

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 1: Deployment Infrastructure

Building code is half the battle. Running it reliably in production is the other half. This chapter teaches you to deploy crypto and Telegram applications on infrastructure that scales, stays online, and handles failures gracefully.

SECTION 1: INFRASTRUCTURE FUNDAMENTALS

Understanding Deployment Architecture

Production deployment requires multiple layers:

Compute Layer:

Where your code runs. Servers, containers, serverless functions.

Network Layer:

How traffic reaches your application. Load balancers, DNS, CDN.

Storage Layer:

Where data persists. Databases, file storage, caches.

Monitoring Layer:

How you know what is happening. Logs, metrics, alerts.

Security Layer:

How you protect everything. Firewalls, encryption, access control.

Infrastructure as Code

Never configure infrastructure manually. Define everything in code:

```
from dataclasses import dataclass, field
from typing import List, Dict, Optional
from enum import Enum
import json
```

```
class Provider(Enum):
    AWS = "aws"
    GCP = "gcp"
    AZURE = "azure"
    DIGITALOCEAN = "digitalocean"
    HETZNER = "hetzner"
```

```
class InstanceSize(Enum):
    SMALL = "small"
    MEDIUM = "medium"
    LARGE = "large"
    XLARGE = "xlarge"
```

```
@dataclass
class ServerConfig:
    name: str
    provider: Provider
    region: str
    size: InstanceSize
    image: str
    ssh_keys: List[str]
    tags: Dict[str, str] = field(default_factory=dict)
    user_data: Optional[str] = None
```

```
@dataclass
class DatabaseConfig:
    name: str
    engine: str
    version: str
    size: InstanceSize
    storage_gb: int
    backup_retention_days: int
    high_availability: bool
    encryption_at_rest: bool
```

```
@dataclass
class NetworkConfig:
    vpc_cidr: str
    public_subnets: List[str]
    private_subnets: List[str]
    nat_gateway: bool
    vpn_enabled: bool
```

```
@dataclass
class InfrastructureSpec:
    name: str
    environment: str
    provider: Provider
    region: str
    network: NetworkConfig
    servers: List[ServerConfig]
    databases: List[DatabaseConfig]
```

```
def to_terraform(self) -> str:
    tf = f"""
terraform {{
    required_providers {{
        {self.provider.value} = {{
            source = "hashicorp/{self.provider.value}"
            version = "~> 4.0"
        }}
    }}
}}

provider "{self.provider.value}" {{
    region = "{self.region}"
}}
"""
    return tf
```

SECTION 2: CONTAINER DEPLOYMENT

Docker Configuration

Package your application in containers for consistent deployment:

FROM python:3.11-slim as base

ENV PYTHONDONTWRITEBYTECODE = 1

ENV PYTHONUNBUFFERED = 1

WORKDIR /app

RUN apt-get update && apt-get install -y \
gcc \
libpq-dev \
&& rm -rf /var/lib/apt/lists/*

FROM base as builder

COPY requirements.txt .

RUN pip wheel --no-cache-dir --no-deps --wheel-dir /wheels -r
requirements.txt

FROM base as production

RUN useradd --create-home --shell /bin/bash appuser

COPY --from = builder /wheels /wheels

RUN pip install --no-cache /wheels/*

COPY --chown = appuser:appuser . .

USER appuser

EXPOSE 8000

HEALTHCHECK --interval = 30s --timeout = 10s --start-period = 5s --
retries = 3 \
CMD python -c "import urllib.request;

urllib.request.urlopen('http://localhost:8000/health')"

CMD ["python", "-m", "gunicorn", "app:create_app()", "-b",

"0.0.0.0:8000", "-w", "4"]

Docker Compose for Local Development

version: '3.8'

services:

app:

build:

context: .

dockerfile: Dockerfile

ports:

- "8000:8000"

environment:

- DATABASE_URL = postgresql://user:pass@db:5432/app

- REDIS_URL = redis://redis:6379/0

- TELEGRAM_BOT_TOKEN = \${TELEGRAM_BOT_TOKEN}

- ETH_RPC_URL = \${ETH_RPC_URL}

depends_on:

db:

condition: service_healthy

redis:

condition: service_healthy

restart: unless-stopped

networks:

- app-network

db:

image: postgres:15-alpine

environment:

- POSTGRES_USER = user

- POSTGRES_PASSWORD = pass

- POSTGRES_DB = app

volumes:

- postgres_data:/var/lib/postgresql/data

healthcheck:

test: ["CMD-SHELL", "pg_isready -U user -d app"]

interval: 10s

timeout: 5s

retries: 5

networks:

- app-network

redis:

image: redis:7-alpine

command: redis-server --appendonly yes

volumes:

- redis_data:/data

healthcheck:

test: ["CMD", "redis-cli", "ping"]

interval: 10s

timeout: 5s

retries: 5

networks:

- app-network

worker:

build:

context: .

dockerfile: Dockerfile

command: python -m celery -A tasks worker --loglevel = info

environment:

- DATABASE_URL = postgresql://user:pass@db:5432/app

- REDIS_URL = redis://redis:6379/0

depends_on:

- db

- redis

restart: unless-stopped

networks:

- app-network

volumes:

postgres_data:

redis_data:

networks:

app-network:

driver: bridge

SECTION 3: KUBERNETES DEPLOYMENT

Kubernetes Manifests

For production scale, use Kubernetes:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: crypto-bot
  labels:
    app: crypto-bot
spec:
  replicas: 3
  selector:
    matchLabels:
      app: crypto-bot
  template:
    metadata:
      labels:
        app: crypto-bot
    spec:
      containers:
        - name: crypto-bot
          image: registry.example.com/crypto-bot:latest
          ports:
            - containerPort: 8000
          envFrom:
            - secretRef:
                name: crypto-bot-secrets
            - configMapRef:
                name: crypto-bot-config
      resources:
        requests:
          memory: "256Mi"
          cpu: "250m"
        limits:
          memory: "512Mi"
          cpu: "500m"
      livenessProbe:
```

httpGet:
path: /health
port: 8000
initialDelaySeconds: 10
periodSeconds: 30
readinessProbe:
httpGet:
path: /ready
port: 8000
initialDelaySeconds: 5
periodSeconds: 10
volumeMounts:
- name: config-volume
mountPath: /app/config
readOnly: true
volumes:
- name: config-volume
configMap:
name: crypto-bot-config-files

apiVersion: v1
kind: Service
metadata:
name: crypto-bot-service
spec:
selector:
app: crypto-bot
ports:
- protocol: TCP
port: 80
targetPort: 8000
type: ClusterIP

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
name: crypto-bot-ingress
annotations:


```
kubernetes.io/ingress.class: nginx
cert-manager.io/cluster-issuer: letsencrypt-prod
spec:
  tls:
  - hosts:
  - api.example.com
    secretName: crypto-bot-tls
  rules:
  - host: api.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: crypto-bot-service
            port:
              number: 80
```

Helm Chart Structure

```
crypto-bot/
Chart.yaml
values.yaml
templates/
deployment.yaml
service.yaml
ingress.yaml
configmap.yaml
secret.yaml
hpa.yaml
```

values.yaml:

```
replicaCount: 3
```

```
image:
  repository: registry.example.com/crypto-bot
  tag: latest
  pullPolicy: Always
```

service:
type: ClusterIP
port: 80
targetPort: 8000

ingress:
enabled: true
className: nginx
hosts:
- host: api.example.com
paths:
- path: /
pathType: Prefix
tls:
- secretName: crypto-bot-tls
hosts:
- api.example.com

resources:
requests:
memory: "256Mi"
cpu: "250m"
limits:
memory: "512Mi"
cpu: "500m"

autoscaling:
enabled: true
minReplicas: 3
maxReplicas: 10
targetCPUUtilizationPercentage: 70
targetMemoryUtilizationPercentage: 80

env:
LOG_LEVEL: "INFO"
ENVIRONMENT: "production"

secrets:
TELEGRAM_BOT_TOKEN: ""

```
DATABASE_URL: ""  
ETH_RPC_URL: ""
```

SECTION 4: DATABASE DEPLOYMENT

PostgreSQL Production Setup

```
from dataclasses import dataclass  
from typing import Optional  
import subprocess
```

```
@dataclass  
class PostgresConfig:  
    host: str  
    port: int  
    database: str  
    user: str  
    password: str  
    ssl_mode: str = "require"  
    pool_size: int = 20  
    max_overflow: int = 10  
    pool_timeout: int = 30  
  
    def connection_string(self) -> str:  
        return (  
            f"postgresql://{self.user}:{self.password}@"  
            f"{self.host}:{self.port}/{self.database}"  
            f"?sslmode={self.ssl_mode}"  
        )
```

```
class DatabaseDeployment:
```

```
    def __init__(self, config: PostgresConfig):  
        self.config = config
```

```
    def run_migrations(self, migrations_path: str) -> bool:  
        cmd = [
```

```
"alembic",  
"-c", f"{migrations_path}/alembic.ini",  
"upgrade", "head"  
]
```

```
result = subprocess.run(  
    cmd,  
    env = {"DATABASE_URL": self.config.connection_string()},  
    capture_output = True,  
    text = True  
)
```

```
return result.returncode == 0
```

```
def create_read_replica_config(self, replica_host: str) ->  
PostgresConfig:  
    return PostgresConfig(  
        host=replica_host,  
        port=self.config.port,  
        database=self.config.database,  
        user=self.config.user,  
        password=self.config.password,  
        ssl_mode=self.config.ssl_mode,  
        pool_size=self.config.pool_size // 2,  
        max_overflow=self.config.max_overflow // 2,  
        pool_timeout=self.config.pool_timeout  
    )
```

Database Connection Pool

```
from contextlib import contextmanager  
from typing import Generator  
import sqlalchemy  
from sqlalchemy.orm import Session, sessionmaker  
from sqlalchemy.pool import QueuePool
```

```
class DatabasePool:
```

```

def __init__(self, config: PostgresConfig):
    self.config = config
    self.engine = self._create_engine()
    self.SessionLocal = sessionmaker(
        autocommit=False,
        autoflush=False,
        bind=self.engine
    )

```

```

def _create_engine(self) -> sqlalchemy.Engine:
    return sqlalchemy.create_engine(
        self.config.connection_string(),
        poolclass=QueuePool,
        pool_size=self.config.pool_size,
        max_overflow=self.config.max_overflow,
        pool_timeout=self.config.pool_timeout,
        pool_pre_ping=True,
        pool_recycle=3600,
        echo=False
    )

```

```

@contextmanager
def get_session(self) -> Generator[Session, None, None]:
    session = self.SessionLocal()
    try:
        yield session
        session.commit()
    except Exception:
        session.rollback()
        raise
    finally:
        session.close()

```

```

def health_check(self) -> bool:
    try:
        with self.engine.connect() as conn:
            conn.execute(sqlalchemy.text("SELECT 1"))
        return True
    except Exception:
        return False

```

SECTION 5: REDIS DEPLOYMENT

Redis Configuration

```
from dataclasses import dataclass
from typing import Optional, List
import redis
from redis.sentinel import Sentinel
```

```
@dataclass
class RedisConfig:
    host: str
    port: int
    password: Optional[str] = None
    db: int = 0
    ssl: bool = True
    socket_timeout: float = 5.0
    connection_pool_size: int = 10
```

```
@dataclass
class RedisSentinelConfig:
    sentinels: List[tuple]
    master_name: str
    password: Optional[str] = None
    db: int = 0
    socket_timeout: float = 5.0
```

```
class RedisDeployment:
```

```
    def __init__(self, config: RedisConfig):
        self.config = config
        self.pool = self._create_pool()
```

```
    def _create_pool(self) -> redis.ConnectionPool:
        return redis.ConnectionPool(
```

```
host = self.config.host,  
port = self.config.port,  
password = self.config.password,  
db = self.config.db,  
socket_timeout = self.config.socket_timeout,  
max_connections = self.config.connection_pool_size,  
decode_responses = True  
)
```

```
def get_client(self) -> redis.Redis:  
return redis.Redis(connection_pool = self.pool)
```

```
def health_check(self) -> bool:  
try:  
    client = self.get_client()  
    return client.ping()  
except Exception:  
    return False
```

```
class RedisSentinelDeployment:
```

```
def __init__(self, config: RedisSentinelConfig):  
    self.config = config  
    self.sentinel = Sentinel(  
        config.sentinel,  
        socket_timeout = config.socket_timeout,  
        password = config.password  
    )
```

```
def get_master(self) -> redis.Redis:  
return self.sentinel.master_for(  
    self.config.master_name,  
    socket_timeout = self.config.socket_timeout,  
    db = self.config.db  
)
```

```
def get_slave(self) -> redis.Redis:  
return self.sentinel.slave_for(  
    self.config.master_name,
```

```
socket_timeout = self.config.socket_timeout,  
db = self.config.db  
)
```

SECTION 6: LOAD BALANCING

Load Balancer Configuration

NGINX configuration for load balancing:

```
upstream crypto_bot_backend {  
    least_conn;  
    server 10.0.1.10:8000 weight=5;  
    server 10.0.1.11:8000 weight=5;  
    server 10.0.1.12:8000 weight=5;  
  
    keepalive 32;  
}  
  
server {  
    listen 80;  
    listen [::]:80;  
    server_name api.example.com;  
    return 301 https://$server_name$request_uri;  
}  
  
server {  
    listen 443 ssl http2;  
    listen [::]:443 ssl http2;  
    server_name api.example.com;  
  
    ssl_certificate /etc/ssl/certs/api.example.com.crt;  
    ssl_certificate_key /etc/ssl/private/api.example.com.key;  
  
    ssl_protocols TLSv1.2 TLSv1.3;  
    ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-  
AES128-GCM-SHA256;  
    ssl_prefer_server_ciphers off;
```



```
ssl_session_timeout 1d;  
ssl_session_cache shared:SSL:50m;  
ssl_session_tickets off;
```

```
ssl_stapling on;  
ssl_stapling_verify on;
```

```
add_header Strict-Transport-Security "max-age=63072000" always;  
add_header X-Frame-Options DENY;  
add_header X-Content-Type-Options nosniff;
```

```
location / {  
    proxy_pass http://crypto_bot_backend;  
    proxy_http_version 1.1;
```

```
    proxy_set_header Host $host;  
    proxy_set_header X-Real-IP $remote_addr;  
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;  
    proxy_set_header X-Forwarded-Proto $scheme;
```

```
    proxy_set_header Connection "";
```

```
    proxy_connect_timeout 60s;  
    proxy_send_timeout 60s;  
    proxy_read_timeout 60s;
```

```
    proxy_buffering on;  
    proxy_buffer_size 4k;  
    proxy_buffers 8 4k;  
}
```

```
location /health {  
    access_log off;  
    return 200 "healthy\n";  
    add_header Content-Type text/plain;  
}
```

```
location /ws {  
    proxy_pass http://crypto_bot_backend;  
    proxy_http_version 1.1;
```

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
proxy_set_header Host $host;
proxy_read_timeout 86400;
}
}
```

AWS Application Load Balancer

```
resource "aws_lb" "crypto_bot" {
  name = "crypto-bot-alb"
  internal = false
  load_balancer_type = "application"
  security_groups = [aws_security_group.alb.id]
  subnets = var.public_subnets
```

```
  enable_deletion_protection = true
```

```
  tags = {
    Environment = var.environment
  }
}
```

```
resource "aws_lb_target_group" "crypto_bot" {
  name = "crypto-bot-tg"
  port = 8000
  protocol = "HTTP"
  vpc_id = var.vpc_id
```

```
  health_check {
    enabled = true
    healthy_threshold = 2
    interval = 30
    matcher = "200"
    path = "/health"
    port = "traffic-port"
    protocol = "HTTP"
    timeout = 5
    unhealthy_threshold = 3
  }
}
```

```

stickiness {
  type = "lb_cookie"
  cookie_duration = 3600
  enabled = true
}
}

resource "aws_lb_listener" "https" {
  load_balancer_arn = aws_lb.crypto_bot.arn
  port = "443"
  protocol = "HTTPS"
  ssl_policy = "ELBSecurityPolicy-TLS13-1-2-2021-06"
  certificate_arn = var.certificate_arn

  default_action {
    type = "forward"
    target_group_arn = aws_lb_target_group.crypto_bot.arn
  }
}

```

SECTION 7: SECRETS MANAGEMENT

HashiCorp Vault Integration

```

from dataclasses import dataclass
from typing import Dict, Optional
import hvac

```

```

@dataclass
class VaultConfig:
    url: str
    token: Optional[str] = None
    role_id: Optional[str] = None
    secret_id: Optional[str] = None
    mount_point: str = "secret"

```

```
class SecretsManager:
```

```
    def __init__(self, config: VaultConfig):  
        self.config = config  
        self.client = self._create_client()
```

```
    def _create_client(self) -> hvac.Client:  
        client = hvac.Client(url=self.config.url)
```

```
        if self.config.token:  
            client.token = self.config.token  
        elif self.config.role_id and self.config.secret_id:  
            response = client.auth.approle.login(  
                role_id=self.config.role_id,  
                secret_id=self.config.secret_id  
            )  
            client.token = response["auth"]["client_token"]
```

```
    return client
```

```
    def get_secret(self, path: str) -> Dict:  
        response = self.client.secrets.kv.v2.read_secret_version(  
            path=path,  
            mount_point=self.config.mount_point  
        )  
        return response["data"]["data"]
```

```
    def set_secret(self, path: str, data: Dict) -> None:  
        self.client.secrets.kv.v2.create_or_update_secret(  
            path=path,  
            secret=data,  
            mount_point=self.config.mount_point  
        )
```

```
    def get_database_credentials(self, role: str) -> Dict:  
        response = self.client.secrets.database.generate_credentials(  
            name=role  
        )  
        return {  
            "username": response["data"]["username"],
```

```
"password": response["data"]["password"]
}
```

```
class EnvironmentSecrets:
```

```
    def __init__(self, secrets_manager: SecretsManager, environment:
str):
```

```
        self.secrets_manager = secrets_manager
```

```
        self.environment = environment
```

```
        self._cache: Dict[str, str] = {}
```

```
    def get(self, key: str) -> str:
```

```
        if key in self._cache:
```

```
            return self._cache[key]
```

```
        path = f'{self.environment}/{key}'
```

```
        secret = self.secrets_manager.get_secret(path)
```

```
        value = secret.get("value", "")
```

```
        self._cache[key] = value
```

```
        return value
```

```
    def get_all_app_secrets(self) -> Dict[str, str]:
```

```
        path = f'{self.environment}/app'
```

```
        return self.secrets_manager.get_secret(path)
```

AWS Secrets Manager

```
import boto3
```

```
import json
```

```
from typing import Dict
```

```
class AWSSecretsManager:
```

```
    def __init__(self, region: str = "us-east-1"):
```

```
        self.client = boto3.client("secretsmanager", region_name=region)
```

```

def get_secret(self, secret_name: str) -> Dict:
    response = self.client.get_secret_value(SecretId=secret_name)

    if "SecretString" in response:
        return json.loads(response["SecretString"])

    return {}

def create_secret(self, secret_name: str, secret_value: Dict) -> str:
    response = self.client.create_secret(
        Name=secret_name,
        SecretString=json.dumps(secret_value)
    )
    return response["ARN"]

def update_secret(self, secret_name: str, secret_value: Dict) ->
None:
    self.client.update_secret(
        SecretId=secret_name,
        SecretString=json.dumps(secret_value)
    )

def rotate_secret(self, secret_name: str) -> None:
    self.client.rotate_secret(SecretId=secret_name)

```

SECTION 8: DEPLOYMENT AUTOMATION

CI/CD Pipeline

```

from dataclasses import dataclass
from typing import List, Optional
from enum import Enum

class DeploymentStage(Enum):
    BUILD = "build"
    TEST = "test"
    SECURITY_SCAN = "security_scan"
    STAGING = "staging"

```

```
PRODUCTION = "production"
```

```
@dataclass
class DeploymentPipeline:
    name: str
    repository: str
    branch: str
    stages: List[DeploymentStage]
    auto_deploy_staging: bool = True
    require_approval_production: bool = True
```

```
class DeploymentAutomation:

    def __init__(self, pipeline: DeploymentPipeline):
        self.pipeline = pipeline

    def generate_github_actions(self) -> str:
        return f"""
name: {self.pipeline.name}

on:
  push:
    branches: [{self.pipeline.branch}]
  pull_request:
    branches: [{self.pipeline.branch}]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{{{ github.repository }}}}}

jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      image_tag: ${{{{ steps.meta.outputs.tags }}}}}

  steps:
    - uses: actions/checkout@v3
```

- name: Set up Docker Buildx
uses: docker/setup-buildx-action@v2

- name: Log in to Container Registry
uses: docker/login-action@v2
with:
registry: \${{{{ env.REGISTRY }}}}
username: \${{{{ github.actor }}}}
password: \${{{{ secrets.GITHUB_TOKEN }}}}

- name: Extract metadata
id: meta
uses: docker/metadata-action@v4
with:
images: \${{{{ env.REGISTRY }}}}/\${{{{{ env.IMAGE_NAME }}}}
tags: |
type = sha,prefix =
type = ref,event = branch
type = semver,pattern = {{{{version}}}}

- name: Build and push
uses: docker/build-push-action@v4
with:
context: .
push: true
tags: \${{{{ steps.meta.outputs.tags }}}}
cache-from: type = gha
cache-to: type = gha,mode = max

test:
needs: build
runs-on: ubuntu-latest

steps:
- uses: actions/checkout@v3

- name: Set up Python
uses: actions/setup-python@v4
with:

python-version: '3.11'

- name: Install dependencies

run: |

pip install -r requirements.txt

pip install pytest pytest-cov

- name: Run tests

run: pytest --cov = app --cov-report = xml

- name: Upload coverage

uses: codecov/codecov-action@v3

security:

needs: build

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v3

- name: Run Trivy vulnerability scanner

uses: aquasecurity/trivy-action@master

with:

image-ref: \${{{{ env.REGISTRY }}}}/\${{{{{ env.IMAGE_NAME }}}}:\${{{{ github.sha }}}}

format: 'sarif'

output: 'trivy-results.sarif'

- name: Upload Trivy scan results

uses: github/codeql-action/upload-sarif@v2

with:

sarif_file: 'trivy-results.sarif'

deploy-staging:

needs: [test, security]

runs-on: ubuntu-latest

if: github.ref == 'refs/heads/{self.pipeline.branch}'

environment: staging

steps:

```

- name: Deploy to staging
uses: appleboy/ssh-action@master
with:
host: ${ secrets.STAGING_HOST }
username: ${ secrets.SSH_USER }
key: ${ secrets.SSH_KEY }
script: |
cd /app
docker-compose pull
docker-compose up -d

deploy-production:
needs: deploy-staging
runs-on: ubuntu-latest
if: github.ref == 'refs/heads/{self.pipeline.branch}'
environment: production

steps:
- name: Deploy to production
uses: appleboy/ssh-action@master
with:
host: ${ secrets.PRODUCTION_HOST }
username: ${ secrets.SSH_USER }
key: ${ secrets.SSH_KEY }
script: |
cd /app
docker-compose pull
docker-compose up -d --scale app=3

```

Blue-Green Deployment

```
class BlueGreenDeployment:
```

```

    def __init__(self, load_balancer_arn: str, region: str = "us-east-1"):
        self.lb_arn = load_balancer_arn
        self.client = boto3.client("elbv2", region_name=region)
        self.ecs_client = boto3.client("ecs", region_name=region)

```

```
def get_current_target_group(self) -> str:
    response = self.client.describe_listeners(
        LoadBalancerArn = self.lb_arn
    )
```

```
    for listener in response["Listeners"]:
        if listener["Port"] == 443:
            return listener["DefaultActions"][0]["TargetGroupArn"]

    return ""
```

```
def switch_target_group(self, listener_arn: str, new_target_group_arn:
    str) -> None:
    self.client.modify_listener(
        ListenerArn = listener_arn,
        DefaultActions = [{
            "Type": "forward",
            "TargetGroupArn": new_target_group_arn
        }]
    )
```

```
def deploy(
    self,
    blue_target_group: str,
    green_target_group: str,
    listener_arn: str
) -> bool:
```

```
    current = self.get_current_target_group()
```

```
    if current == blue_target_group:
        new_target = green_target_group
    else:
        new_target = blue_target_group
```

```
    if not self.health_check(new_target):
        return False
```

```
    self.switch_target_group(listener_arn, new_target)
```

```
return True
```

```
def health_check(self, target_group_arn: str) -> bool:  
    response = self.client.describe_target_health(  
        TargetGroupArn=target_group_arn  
    )
```

```
    healthy_count = sum(  
        1 for t in response["TargetHealthDescriptions"]  
        if t["TargetHealth"]["State"] == "healthy"  
    )
```

```
    return healthy_count >= 2
```

```
def rollback(self, listener_arn: str, previous_target_group: str) ->  
None:  
    self.switch_target_group(listener_arn, previous_target_group)
```

This chapter covered deployment infrastructure fundamentals. The next chapter covers monitoring and observability for production systems.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 2: Monitoring and Observability

You cannot fix what you cannot see. Production systems require comprehensive monitoring to detect problems before users do. This chapter teaches you to build observability into your crypto and Telegram applications from the ground up.

SECTION 1: OBSERVABILITY FUNDAMENTALS

The Three Pillars

Observability rests on three pillars:

Metrics:

Numeric measurements over time. CPU usage. Request count. Error rate. Response latency.

Logs:

Timestamped records of discrete events. User actions. Errors. State changes.

Traces:

Request flow across services. How long each step takes. Where bottlenecks occur.

Together they answer: What is happening? Why is it happening? Where is it happening?

Why Observability Matters for Crypto

Crypto applications have unique monitoring needs:

Financial Transactions:

Every transaction involves real money. Missing or delayed transactions means user funds at risk. You need immediate visibility into transaction status.

Blockchain State:

Your application depends on blockchain data. Node health. Block synchronization. Gas prices. Network congestion.

Bot Availability:

Telegram bots must respond quickly. Users expect sub-second responses. Downtime means lost trades and frustrated users.

Security Events:

Unusual patterns may indicate attacks. Large withdrawals. Repeated failed auth. Address changes.

SECTION 2: METRICS COLLECTION

Prometheus Metrics

Prometheus is the standard for metrics collection:

```
from prometheus_client import Counter, Histogram, Gauge, Info
from prometheus_client import generate_latest,
CONTENT_TYPE_LATEST
from functools import wraps
import time
```

```
REQUESTS_TOTAL = Counter(
    "http_requests_total",
    "Total HTTP requests",
    ["method", "endpoint", "status"]
)
```

```
REQUEST_DURATION = Histogram(
    "http_request_duration_seconds",
    "HTTP request duration in seconds",
    ["method", "endpoint"],
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0, 10.0]
)
```

```
ACTIVE_CONNECTIONS = Gauge(
    "active_connections",
    "Number of active connections"
)
```

```
APP_INFO = Info(
    "app",
    "Application information"
)
```

```
class MetricsCollector:
```

```
    def __init__(self):
        self.custom_metrics = {}
```

```
    def track_request(self, method: str, endpoint: str, status: int,
        duration: float):
        REQUESTS_TOTAL.labels(method=method, endpoint=endpoint,
        status=status).inc()
```

```
REQUEST_DURATION.labels(method = method,  
endpoint = endpoint).observe(duration)
```

```
def set_connection_count(self, count: int):  
    ACTIVE_CONNECTIONS.set(count)
```

```
def set_app_info(self, version: str, environment: str):  
    APP_INFO.info({  
        "version": version,  
        "environment": environment  
    })
```

```
def create_counter(self, name: str, description: str, labels: list) ->  
Counter:  
    counter = Counter(name, description, labels)  
    self.custom_metrics[name] = counter  
    return counter
```

```
def create_histogram(  
    self,  
    name: str,  
    description: str,  
    labels: list,  
    buckets: list = None  
) -> Histogram:
```

```
    if buckets is None:  
        buckets = [0.1, 0.5, 1.0, 2.5, 5.0, 10.0]
```

```
    histogram = Histogram(name, description, labels,  
buckets=buckets)  
    self.custom_metrics[name] = histogram  
    return histogram
```

```
def create_gauge(self, name: str, description: str, labels: list =  
None) -> Gauge:  
    if labels:  
        gauge = Gauge(name, description, labels)  
    else:  
        gauge = Gauge(name, description)
```

```
self.custom_metrics[name] = gauge
return gauge
```

```
def track_time(histogram: Histogram, labels: dict = None):
    def decorator(func):
        @wraps(func)
        async def wrapper(*args, **kwargs):
            start = time.time()
            try:
                return await func(*args, **kwargs)
            finally:
                duration = time.time() - start
                if labels:
                    histogram.labels(**labels).observe(duration)
                else:
                    histogram.observe(duration)
            return wrapper
        return decorator
```

Crypto-Specific Metrics

```
class CryptoMetrics:
```

```
    def __init__(self):
        self.transactions_sent = Counter(
            "blockchain_transactions_sent_total",
            "Total blockchain transactions sent",
            ["chain", "type", "status"]
        )

        self.transaction_gas = Histogram(
            "blockchain_transaction_gas_used",
            "Gas used per transaction",
            ["chain", "type"],
            buckets=[21000, 50000, 100000, 200000, 500000, 1000000]
        )
```



```
self.transaction_value = Histogram(  
    "blockchain_transaction_value_wei",  
    "Transaction value in wei",  
    ["chain", "type"],  
    buckets=[1e15, 1e16, 1e17, 1e18, 1e19, 1e20]  
)
```

```
self.wallet_balance = Gauge(  
    "wallet_balance_wei",  
    "Wallet balance in wei",  
    ["address", "chain"]  
)
```

```
self.gas_price = Gauge(  
    "blockchain_gas_price_gwei",  
    "Current gas price in gwei",  
    ["chain"]  
)
```

```
self.block_number = Gauge(  
    "blockchain_block_number",  
    "Current block number",  
    ["chain"]  
)
```

```
self.node_latency = Histogram(  
    "blockchain_node_latency_seconds",  
    "RPC node response latency",  
    ["chain", "method"],  
    buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0]  
)
```

```
self.pending_transactions = Gauge(  
    "blockchain_pending_transactions",  
    "Number of pending transactions",  
    ["chain"]  
)
```

```
def record_transaction(  
    self,
```

```

chain: str,
tx_type: str,
status: str,
gas_used: int,
value: int
):
    self.transactions_sent.labels(chain = chain, type = tx_type,
status = status).inc()
    self.transaction_gas.labels(chain = chain,
type = tx_type).observe(gas_used)
    self.transaction_value.labels(chain = chain,
type = tx_type).observe(value)

def update_wallet_balance(self, address: str, chain: str, balance: int):
    self.wallet_balance.labels(address = address,
chain = chain).set(balance)

def update_gas_price(self, chain: str, price_gwei: float):
    self.gas_price.labels(chain = chain).set(price_gwei)

def update_block_number(self, chain: str, block: int):
    self.block_number.labels(chain = chain).set(block)

```

```

class TelegramMetrics:

```

```

    def __init__(self):
        self.messages_received = Counter(
            "telegram_messages_received_total",
            "Total Telegram messages received",
            ["chat_type", "message_type"]
        )

```

```

        self.messages_sent = Counter(
            "telegram_messages_sent_total",
            "Total Telegram messages sent",
            ["chat_type", "message_type"]
        )

```

```

        self.command_executions = Counter(

```

```
"telegram_command_executions_total",  
"Total command executions",  
["command", "status"]  
)
```

```
self.callback_queries = Counter(  
"telegram_callback_queries_total",  
"Total callback queries",  
["callback_data"]  
)
```

```
self.message_latency = Histogram(  
"telegram_message_processing_seconds",  
"Message processing time",  
["message_type"],  
buckets=[0.01, 0.05, 0.1, 0.25, 0.5, 1.0, 2.5, 5.0]  
)
```

```
self.active_users = Gauge(  
"telegram_active_users",  
"Number of active users in last 24h"  
)
```

```
self.webhook_requests = Counter(  
"telegram_webhook_requests_total",  
"Total webhook requests",  
["status"]  
)
```

```
def record_message(self, chat_type: str, message_type: str,  
processing_time: float):  
    self.messages_received.labels(chat_type=chat_type,  
message_type=message_type).inc()  
    self.message_latency.labels(message_type=message_type).observe(processing_time)  
  
def record_command(self, command: str, status: str):  
    self.command_executions.labels(command=command,  
status=status).inc()
```

SECTION 3: STRUCTURED LOGGING

Logging Configuration

```
import logging
import json
import sys
from datetime import datetime
from typing import Any, Dict, Optional
import traceback

class JSONFormatter(logging.Formatter):

    def __init__(self, service_name: str, environment: str):
        super().__init__()
        self.service_name = service_name
        self.environment = environment

    def format(self, record: logging.LogRecord) -> str:
        log_entry = {
            "timestamp": datetime.utcnow().isoformat() + "Z",
            "level": record.levelname,
            "logger": record.name,
            "message": record.getMessage(),
            "service": self.service_name,
            "environment": self.environment
        }

        if record.exc_info:
            log_entry["exception"] = {
                "type": record.exc_info[0].__name__ if record.exc_info[0] else None,
                "message": str(record.exc_info[1]) if record.exc_info[1] else None,
                "traceback": traceback.format_exception(*record.exc_info)
            }

        if hasattr(record, "extra_fields"):
            log_entry.update(record.extra_fields)

        return json.dumps(log_entry)
```

```
class StructuredLogger:
```

```
    def __init__(self, name: str, service_name: str, environment: str):
        self.logger = logging.getLogger(name)
        self.logger.setLevel(logging.DEBUG)
```

```
        handler = logging.StreamHandler(sys.stdout)
        handler.setFormatter(JSONFormatter(service_name, environment))
        self.logger.addHandler(handler)
```

```
        self.context: Dict[str, Any] = {}
```

```
    def set_context(self, **kwargs):
        self.context.update(kwargs)
```

```
    def clear_context(self):
        self.context.clear()
```

```
    def _log(self, level: int, message: str, **kwargs):
        extra = {"extra_fields": {**self.context, **kwargs}}
        self.logger.log(level, message, extra = extra)
```

```
    def debug(self, message: str, **kwargs):
        self._log(logging.DEBUG, message, **kwargs)
```

```
    def info(self, message: str, **kwargs):
        self._log(logging.INFO, message, **kwargs)
```

```
    def warning(self, message: str, **kwargs):
        self._log(logging.WARNING, message, **kwargs)
```

```
    def error(self, message: str, **kwargs):
        self._log(logging.ERROR, message, **kwargs)
```

```
    def critical(self, message: str, **kwargs):
        self._log(logging.CRITICAL, message, **kwargs)
```

```
    def exception(self, message: str, **kwargs):
```

```
self._log(logging.ERROR, message, exc_info = True, **kwargs)
```

Application-Specific Logging

```
class TransactionLogger:
```

```
    def __init__(self, base_logger: StructuredLogger):
        self.logger = base_logger
```

```
    def log_transaction_initiated(
        self,
        tx_id: str,
        from_address: str,
        to_address: str,
        value: int,
        chain: str
    ):
        self.logger.info(
            "Transaction initiated",
            event="tx_initiated",
            tx_id=tx_id,
            from_address=from_address,
            to_address=to_address,
            value_wei=value,
            chain=chain
        )
```

```
    def log_transaction_submitted(
        self,
        tx_id: str,
        tx_hash: str,
        gas_price: int,
        nonce: int
    ):
        self.logger.info(
            "Transaction submitted to network",
            event="tx_submitted",
            tx_id=tx_id,
            tx_hash=tx_hash,
```

```
gas_price_gwei = gas_price / 1e9,  
nonce = nonce  
)
```

```
def log_transaction_confirmed(  
self,  
tx_id: str,  
tx_hash: str,  
block_number: int,  
gas_used: int,  
status: str  
):  
self.logger.info(  
"Transaction confirmed",  
event = "tx_confirmed",  
tx_id = tx_id,  
tx_hash = tx_hash,  
block_number = block_number,  
gas_used = gas_used,  
status = status  
)
```

```
def log_transaction_failed(  
self,  
tx_id: str,  
tx_hash: Optional[str],  
error: str,  
error_code: Optional[str]  
):  
self.logger.error(  
"Transaction failed",  
event = "tx_failed",  
tx_id = tx_id,  
tx_hash = tx_hash,  
error = error,  
error_code = error_code  
)
```

```
class TelegramLogger:
```

```
def __init__(self, base_logger: StructuredLogger):  
    self.logger = base_logger
```

```
def log_message_received(  
    self,  
    message_id: int,  
    chat_id: int,  
    user_id: int,  
    message_type: str,  
    text_preview: Optional[str] = None  
):  
    self.logger.info(  
        "Telegram message received",  
        event="telegram_message_received",  
        message_id=message_id,  
        chat_id=chat_id,  
        user_id=user_id,  
        message_type=message_type,  
        text_preview=text_preview[:50] if text_preview else None  
    )
```

```
def log_command_executed(  
    self,  
    command: str,  
    user_id: int,  
    chat_id: int,  
    duration_ms: float,  
    success: bool  
):  
    self.logger.info(  
        f"Command executed: {command}",  
        event="telegram_command_executed",  
        command=command,  
        user_id=user_id,  
        chat_id=chat_id,  
        duration_ms=duration_ms,  
        success=success  
    )
```



```

def log_rate_limit_exceeded(
    self,
    user_id: int,
    limit_type: str,
    current_count: int,
    limit: int
):
    self.logger.warning(
        "Rate limit exceeded",
        event="telegram_rate_limit",
        user_id=user_id,
        limit_type=limit_type,
        current_count=current_count,
        limit=limit
    )

```

SECTION 4: DISTRIBUTED TRACING

OpenTelemetry Integration

```

from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter import
OTLPSpanExporter
from opentelemetry.sdk.resources import Resource
from opentelemetry.instrumentation.requests import
RequestsInstrumentor
from opentelemetry.instrumentation.sqlalchemy import
SQLAlchemyInstrumentor
from opentelemetry.propagate import set_global_textmap
from opentelemetry.propagators.b3 import B3MultiFormat
from contextlib import contextmanager
from typing import Optional, Dict, Any

```

```

class TracingSetup:

```

```

    def __init__(

```

```

self,
service_name: str,
environment: str,
otlp_endpoint: str
):
self.service_name = service_name
self.environment = environment
self.otlp_endpoint = otlp_endpoint

self._setup_tracing()

def _setup_tracing(self):
resource = Resource.create({
"service.name": self.service_name,
"deployment.environment": self.environment
})

provider = TracerProvider(resource=resource)

exporter = OTLPSpanExporter(endpoint=self.otlp_endpoint)
processor = BatchSpanProcessor(exporter)
provider.add_span_processor(processor)

trace.set_tracer_provider(provider)

set_global_textmap(B3MultiFormat())

RequestsInstrumentor().instrument()

def instrument_sqlalchemy(self, engine):
SQLAlchemyInstrumentor().instrument(engine=engine)

def get_tracer(self, name: str) -> trace.Tracer:
return trace.get_tracer(name)

class TransactionTracer:

def __init__(self, tracer: trace.Tracer):
self.tracer = tracer

```

```

@contextmanager
def trace_transaction(
    self,
    tx_id: str,
    tx_type: str,
    attributes: Dict[str, Any] = None
):
    with self.tracer.start_as_current_span(
        f"blockchain.transaction.{tx_type}",
        attributes = {
            "tx.id": tx_id,
            "tx.type": tx_type,
            **(attributes or {})
        }
    ) as span:
        try:
            yield span
        except Exception as e:
            span.set_status(trace.Status(trace.StatusCode.ERROR, str(e)))
            span.record_exception(e)
            raise

```

```

@contextmanager
def trace_rpc_call(
    self,
    method: str,
    chain: str,
    params: Optional[Dict] = None
):
    with self.tracer.start_as_current_span(
        f"blockchain.rpc.{method}",
        attributes = {
            "rpc.method": method,
            "blockchain.chain": chain
        }
    ) as span:
        try:
            yield span
        except Exception as e:

```

```

span.set_status(trace.Status(trace.StatusCode.ERROR, str(e)))
raise

@contextmanager
def trace_telegram_handler(
    self,
    handler_name: str,
    user_id: int,
    chat_id: int
):
    with self.tracer.start_as_current_span(
        f'telegram.handler.{handler_name}',
        attributes = {
            "telegram.user_id": user_id,
            "telegram.chat_id": chat_id,
            "telegram.handler": handler_name
        }
    ) as span:
        try:
            yield span
        except Exception as e:
            span.set_status(trace.Status(trace.StatusCode.ERROR, str(e)))
            raise

```

SECTION 5: ALERTING

Alert Configuration

```

from dataclasses import dataclass
from typing import List, Optional, Callable
from enum import Enum
import asyncio
import aiohttp

class AlertSeverity(Enum):
    INFO = "info"
    WARNING = "warning"
    ERROR = "error"

```

```
CRITICAL = "critical"
```

```
@dataclass
class AlertRule:
    name: str
    query: str
    threshold: float
    comparison: str
    duration: str
    severity: AlertSeverity
    message_template: str
    labels: dict = None
```

```
class AlertManager:
```

```
    def __init__(self):
        self.rules: List[AlertRule] = []
        self.channels: List[Callable] = []
        self.active_alerts: dict = {}
```

```
    def add_rule(self, rule: AlertRule):
        self.rules.append(rule)
```

```
    def add_channel(self, channel: Callable):
        self.channels.append(channel)
```

```
    async def evaluate_rules(self, metrics: dict):
        for rule in self.rules:
            value = metrics.get(rule.query, 0)
            triggered = self._check_threshold(value, rule.threshold,
            rule.comparison)
```

```
        if triggered:
            await self._fire_alert(rule, value)
        elif rule.name in self.active_alerts:
            await self._resolve_alert(rule)
```

```
    def _check_threshold(self, value: float, threshold: float, comparison:
```

```
str) -> bool:
```

```
if comparison == ">":  
    return value > threshold  
elif comparison == "<":  
    return value < threshold  
elif comparison == ">=":  
    return value >= threshold  
elif comparison == "<=":  
    return value <= threshold  
elif comparison == "==":  
    return value == threshold  
return False
```

```
async def _fire_alert(self, rule: AlertRule, value: float):
```

```
if rule.name not in self.active_alerts:
```

```
self.active_alerts[rule.name] = {
```

```
"rule": rule,
```

```
"value": value,
```

```
"fired_at": asyncio.get_event_loop().time()
```

```
}
```

```
message = rule.message_template.format(
```

```
name=rule.name,
```

```
value=value,
```

```
threshold=rule.threshold
```

```
)
```

```
for channel in self.channels:
```

```
await channel(rule.severity, rule.name, message)
```

```
async def _resolve_alert(self, rule: AlertRule):
```

```
if rule.name in self.active_alerts:
```

```
del self.active_alerts[rule.name]
```

```
for channel in self.channels:
```

```
await channel(
```

```
AlertSeverity.INFO,
```

```
f'{rule.name} (Resolved)',
```

```
f'Alert {rule.name} has been resolved'
```

```
)
```

```
class AlertChannels:
```

```
    @staticmethod
    async def telegram_channel(
        bot_token: str,
        chat_id: int
    ) -> Callable:
```

```
        async def send(severity: AlertSeverity, title: str, message: str):
            emoji = {
                AlertSeverity.INFO: "i",
                AlertSeverity.WARNING: "!",
                AlertSeverity.ERROR: "X",
                AlertSeverity.CRITICAL: "!!!"
            }
```

```
            text = f'{emoji.get(severity, "")} {severity.value.upper()}: {title}\n
            \n{message}'
```

```
            url = f'https://api.telegram.org/bot{bot_token}/sendMessage'
            async with aiohttp.ClientSession() as session:
                await session.post(url, json = {
                    "chat_id": chat_id,
                    "text": text,
                    "parse_mode": "HTML"
                })
```

```
        return send
```

```
    @staticmethod
    async def slack_channel(webhook_url: str) -> Callable:
```

```
        async def send(severity: AlertSeverity, title: str, message: str):
            color = {
                AlertSeverity.INFO: "#36a64f",
                AlertSeverity.WARNING: "#ffcc00",
                AlertSeverity.ERROR: "#ff6600",
                AlertSeverity.CRITICAL: "#ff0000"
```

```
}
```

```
payload = {  
    "attachments": [{  
        "color": color.get(severity, "#808080"),  
        "title": f"[{severity.value.upper()}] {title}",  
        "text": message,  
        "footer": "Alert Manager"  
    }]  
}
```

```
async with aiohttp.ClientSession() as session:  
    await session.post(webhook_url, json=payload)
```

```
return send
```

```
@staticmethod
```

```
async def pagerduty_channel(routing_key: str) -> Callable:
```

```
async def send(severity: AlertSeverity, title: str, message: str):  
    pd_severity = {  
        AlertSeverity.INFO: "info",  
        AlertSeverity.WARNING: "warning",  
        AlertSeverity.ERROR: "error",  
        AlertSeverity.CRITICAL: "critical"  
    }
```

```
payload = {  
    "routing_key": routing_key,  
    "event_action": "trigger",  
    "payload": {  
        "summary": title,  
        "severity": pd_severity.get(severity, "info"),  
        "source": "crypto-bot",  
        "custom_details": {"message": message}  
    }  
}
```

```
async with aiohttp.ClientSession() as session:  
    await session.post(
```



```
"https://events.pagerduty.com/v2/enqueue",  
json = payload  
)
```

```
return send
```

Default Alert Rules

```
def create_default_alerts() -> List[AlertRule]:  
  return [  
    AlertRule(  
      name = "high_error_rate",  
      query = "error_rate_5m",  
      threshold = 0.05,  
      comparison = ">",  
      duration = "5m",  
      severity = AlertSeverity.ERROR,  
      message_template = "Error rate is {value:.2%}, threshold is  
{threshold:.2%}"  
    ),  
    AlertRule(  
      name = "high_latency",  
      query = "p99_latency_ms",  
      threshold = 5000,  
      comparison = ">",  
      duration = "5m",  
      severity = AlertSeverity.WARNING,  
      message_template = "P99 latency is {value}ms, threshold is  
{threshold}ms"  
    ),  
    AlertRule(  
      name = "low_wallet_balance",  
      query = "wallet_balance_eth",  
      threshold = 0.1,  
      comparison = "<",  
      duration = "1m",  
      severity = AlertSeverity.CRITICAL,  
      message_template = "Wallet balance is {value} ETH, below  
minimum {threshold} ETH"
```

```
),
AlertRule(
name="node_out_of_sync",
query="block_lag",
threshold=10,
comparison=">",
duration="5m",
severity=AlertSeverity.ERROR,
message_template="Node is {value} blocks behind, threshold is {threshold}"
),
AlertRule(
name="high_gas_price",
query="gas_price_gwei",
threshold=100,
comparison=">",
duration="10m",
severity=AlertSeverity.WARNING,
message_template="Gas price is {value} gwei, above threshold {threshold} gwei"
),
AlertRule(
name="pending_transactions_stuck",
query="pending_tx_age_minutes",
threshold=30,
comparison=">",
duration="5m",
severity=AlertSeverity.ERROR,
message_template="Transactions pending for {value} minutes, threshold is {threshold}"
),
AlertRule(
name="telegram_webhook_failures",
query="webhook_failure_rate_5m",
threshold=0.1,
comparison=">",
duration="5m",
severity=AlertSeverity.ERROR,
message_template="Webhook failure rate is {value:.2%}, threshold is {threshold:.2%}"
)
```

```

),
AlertRule(
    name="database_connection_pool_exhausted",
    query="db_pool_available",
    threshold=2,
    comparison="<",
    duration="1m",
    severity=AlertSeverity.CRITICAL,
    message_template="Only {value} DB connections available,
minimum is {threshold}"
)
]

```

SECTION 6: DASHBOARDS

Grafana Dashboard Configuration

```

import json
from typing import List, Dict, Any

class GrafanaDashboard:

    def __init__(self, title: str, uid: str):
        self.title = title
        self.uid = uid
        self.panels: List[Dict] = []
        self.variables: List[Dict] = []
        self.panel_id = 1

    def add_variable(self, name: str, query: str, datasource: str):
        self.variables.append({
            "name": name,
            "type": "query",
            "datasource": datasource,
            "query": query,
            "refresh": 1,
            "multi": False
        })

```

```

def add_stat_panel(
    self,
    title: str,
    query: str,
    x: int,
    y: int,
    width: int = 4,
    height: int = 4
):
    panel = {
        "id": self.panel_id,
        "title": title,
        "type": "stat",
        "gridPos": {"x": x, "y": y, "w": width, "h": height},
        "targets": [{
            "expr": query,
            "refId": "A"
        }],
        "options": {
            "reduceOptions": {
                "calcs": ["lastNotNull"],
                "fields": "",
                "values": False
            },
            "colorMode": "value",
            "graphMode": "area"
        }
    }
    self.panels.append(panel)
    self.panel_id += 1

```

```

def add_graph_panel(
    self,
    title: str,
    queries: List[Dict[str, str]],
    x: int,
    y: int,
    width: int = 12,
    height: int = 8

```

```

):
targets = []
for i, q in enumerate(queries):
targets.append({
"expr": ql["query"],
"legendFormat": q.get("legend", ""),
"refId": chr(65 + i)
})

panel = {
"id": self.panel_id,
"title": title,
"type": "timeseries",
"gridPos": {"x": x, "y": y, "w": width, "h": height},
"targets": targets,
"options": {
"legend": {"displayMode": "list", "placement": "bottom"},
"tooltip": {"mode": "multi"}
},
"fieldConfig": {
"defaults": {
"custom": {
"drawStyle": "line",
"lineWidth": 1,
"fillOpacity": 10
}
}
}
}

self.panels.append(panel)
self.panel_id += 1

def add_table_panel(
self,
title: str,
query: str,
columns: List[str],
x: int,
y: int,
width: int = 12,

```

```

height: int = 8
):
panel = {
    "id": self.panel_id,
    "title": title,
    "type": "table",
    "gridPos": {"x": x, "y": y, "w": width, "h": height},
    "targets": [{
        "expr": query,
        "format": "table",
        "instant": True,
        "refId": "A"
    }],
    "transformations": [{
        "id": "organize",
        "options": {"excludeByName": {}, "indexByName": {}}
    }]
}
self.panels.append(panel)
self.panel_id += 1

```

```

def to_json(self) -> str:
    dashboard = {
        "uid": self.uid,
        "title": self.title,
        "tags": ["crypto", "telegram", "bot"],
        "timezone": "utc",
        "schemaVersion": 38,
        "version": 1,
        "refresh": "30s",
        "time": {"from": "now-1h", "to": "now"},
        "templating": {"list": self.variables},
        "panels": self.panels
    }
    return json.dumps(dashboard, indent=2)

```

```

def create_crypto_bot_dashboard() -> GrafanaDashboard:
    dashboard = GrafanaDashboard("Crypto Bot Overview", "crypto-
bot-main")

```

```
dashboard.add_stat_panel(  
"Active Users (24h)",  
"telegram_active_users",  
0, 0, 4, 4  
)
```

```
dashboard.add_stat_panel(  
"Transactions Today",  
"sum(increase(blockchain_transactions_sent_total[24h]))",  
4, 0, 4, 4  
)
```

```
dashboard.add_stat_panel(  
"Error Rate",  
"sum(rate(http_requests_total{status = ~\"5..\"}[5m])) /  
sum(rate(http_requests_total[5m]))",  
8, 0, 4, 4  
)
```

```
dashboard.add_stat_panel(  
"Current Gas Price",  
"blockchain_gas_price_gwei",  
12, 0, 4, 4  
)
```

```
dashboard.add_graph_panel(  
"Request Rate",  
[  
{"query": "sum(rate(http_requests_total[5m]))", "legend": "Total"},  
{"query": "sum(rate(http_requests_total{status = \"200\"}[5m]))",  
"legend": "Success"},  
{"query": "sum(rate(http_requests_total{status = ~\"5..\"}[5m]))",  
"legend": "Errors"}  
],  
0, 4, 12, 8  
)
```

```
dashboard.add_graph_panel(  
"Response Latency",
```

```
[
  {"query": "histogram_quantile(0.50,
rate(http_request_duration_seconds_bucket[5m]))", "legend": "p50"},
  {"query": "histogram_quantile(0.95,
rate(http_request_duration_seconds_bucket[5m]))", "legend": "p95"},
  {"query": "histogram_quantile(0.99,
rate(http_request_duration_seconds_bucket[5m]))", "legend": "p99"}
],
12, 4, 12, 8
)
```

```
dashboard.add_graph_panel(
  "Telegram Messages",
  [
    {"query": "sum(rate(telegram_messages_received_total[5m])) * 60",
"legend": "Received/min"},
    {"query": "sum(rate(telegram_messages_sent_total[5m])) * 60",
"legend": "Sent/min"}
  ],
  0, 12, 12, 8
)
```

```
dashboard.add_graph_panel(
  "Blockchain Transactions",
  [
    {"query": "sum(rate(blockchain_transactions_sent_total{status =
\"success\"}[5m])) * 60", "legend": "Success"},
    {"query": "sum(rate(blockchain_transactions_sent_total{status =
\"failed\"}[5m])) * 60", "legend": "Failed"},
    {"query": "sum(rate(blockchain_transactions_sent_total{status =
\"pending\"}[5m])) * 60", "legend": "Pending"}
  ],
  12, 12, 12, 8
)
```

```
return dashboard
```

SECTION 7: HEALTH CHECKS

Comprehensive Health Check System

```
from dataclasses import dataclass
from typing import Dict, List, Optional, Callable, Awaitable
from enum import Enum
import asyncio
import time
```

```
class HealthStatus(Enum):
    HEALTHY = "healthy"
    DEGRADED = "degraded"
    UNHEALTHY = "unhealthy"
```

```
@dataclass
class HealthCheckResult:
    name: str
    status: HealthStatus
    message: str
    latency_ms: float
    details: Optional[Dict] = None
```

```
class HealthChecker:

    def __init__(self):
        self.checks: Dict[str, Callable[[], Awaitable[HealthCheckResult]]]
        = {}
```

```
    def register(self, name: str, check: Callable[[],
        Awaitable[HealthCheckResult]]):
        self.checks[name] = check
```

```
    async def run_all(self) -> Dict[str, HealthCheckResult]:
        results = {}
```

```
        tasks = [
            self._run_check(name, check)
            for name, check in self.checks.items()
        ]
```

```
]
```

```
completed = await asyncio.gather(*tasks,  
return_exceptions=True)
```

```
for name, result in zip(self.checks.keys(), completed):  
    if isinstance(result, Exception):  
        results[name] = HealthCheckResult(  
            name=name,  
            status=HealthStatus.UNHEALTHY,  
            message=str(result),  
            latency_ms=0  
        )  
    else:  
        results[name] = result
```

```
return results
```

```
async def _run_check(  
    self,  
    name: str,  
    check: Callable  
) -> HealthCheckResult:
```

```
    start = time.time()  
    try:  
        result = await asyncio.wait_for(check(), timeout=10.0)  
        result.latency_ms = (time.time() - start) * 1000  
        return result  
    except asyncio.TimeoutError:  
        return HealthCheckResult(  
            name=name,  
            status=HealthStatus.UNHEALTHY,  
            message="Health check timed out",  
            latency_ms=10000  
        )
```

```
def overall_status(self, results: Dict[str, HealthCheckResult]) ->  
HealthStatus:  
    statuses = [r.status for r in results.values()]
```

```
if HealthStatus.UNHEALTHY in statuses:
    return HealthStatus.UNHEALTHY
if HealthStatus.DEGRADED in statuses:
    return HealthStatus.DEGRADED
return HealthStatus.HEALTHY
```

```
class StandardHealthChecks:
```

```
    @staticmethod
    async def database_check(pool) -> HealthCheckResult:
        try:
            async with pool.acquire() as conn:
                await conn.execute("SELECT 1")

            return HealthCheckResult(
                name="database",
                status=HealthStatus.HEALTHY,
                message="Database connection OK",
                latency_ms=0
            )
        except Exception as e:
            return HealthCheckResult(
                name="database",
                status=HealthStatus.UNHEALTHY,
                message=f"Database error: {e}",
                latency_ms=0
            )
```

```
    @staticmethod
    async def redis_check(redis_client) -> HealthCheckResult:
        try:
            await redis_client.ping()

            return HealthCheckResult(
                name="redis",
                status=HealthStatus.HEALTHY,
                message="Redis connection OK",
                latency_ms=0
            )
```

```

)
except Exception as e:
    return HealthCheckResult(
        name="redis",
        status=HealthStatus.UNHEALTHY,
        message=f"Redis error: {e}",
        latency_ms=0
    )

```

```

@staticmethod
async def blockchain_node_check(web3) -> HealthCheckResult:
    try:
        block = await web3.eth.block_number
        syncing = await web3.eth.syncing

```

```

    if syncing:
        return HealthCheckResult(
            name="blockchain_node",
            status=HealthStatus.DEGRADED,
            message=f"Node syncing at block {block}",
            latency_ms=0,
            details={"block": block, "syncing": True}
        )

```

```

    return HealthCheckResult(
        name="blockchain_node",
        status=HealthStatus.HEALTHY,
        message=f"Node synced at block {block}",
        latency_ms=0,
        details={"block": block, "syncing": False}
    )

```

```

    except Exception as e:
        return HealthCheckResult(
            name="blockchain_node",
            status=HealthStatus.UNHEALTHY,
            message=f"Node error: {e}",
            latency_ms=0
        )

```

```

@staticmethod

```

```

async def telegram_api_check(bot_token: str) ->
HealthCheckResult:
import aiohttp

try:
url = f"https://api.telegram.org/bot{bot_token}/getMe"
async with aiohttp.ClientSession() as session:
async with session.get(url) as resp:
data = await resp.json()

if data.get("ok"):
return HealthCheckResult(
name="telegram_api",
status=HealthStatus.HEALTHY,
message="Telegram API accessible",
latency_ms=0
)
else:
return HealthCheckResult(
name="telegram_api",
status=HealthStatus.UNHEALTHY,
message=data.get("description", "Unknown error"),
latency_ms=0
)
except Exception as e:
return HealthCheckResult(
name="telegram_api",
status=HealthStatus.UNHEALTHY,
message=f"Telegram API error: {e}",
latency_ms=0
)

```

This chapter covered monitoring and observability for production crypto and Telegram applications. The next chapter covers high availability and disaster recovery.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 3: High Availability

Downtime costs money. For crypto applications, downtime can

mean missed trades, stuck transactions, and lost user trust. This chapter teaches you to build systems that stay online even when components fail.

SECTION 1: HIGH AVAILABILITY CONCEPTS

Understanding Availability

Availability measures uptime:

99% = 3.65 days downtime per year

99.9% = 8.76 hours downtime per year

99.99% = 52.6 minutes downtime per year

99.999% = 5.26 minutes downtime per year

Each additional nine is exponentially harder to achieve. Most production systems target 99.9% to 99.99%.

Failure Modes

Systems fail in many ways:

Hardware Failure:

Servers die. Disks fail. Network cards break. Power supplies burn out.

Software Failure:

Bugs crash processes. Memory leaks exhaust resources. Deadlocks freeze systems.

Network Failure:

Connections drop. DNS fails. Routing breaks. Latency spikes.

Human Error:

Wrong commands. Bad deployments. Misconfiguration. Deleted data.

External Dependencies:

Third-party APIs go down. Blockchain nodes disconnect. Cloud

providers have outages.

High availability means surviving all these failures without user-visible impact.

SECTION 2: REDUNDANCY

Multi-Server Deployment

Never run a single instance of anything critical:

```
from dataclasses import dataclass
from typing import List, Dict, Optional
import asyncio
import random
```

```
@dataclass
class ServerInstance:
    id: str
    host: str
    port: int
    weight: int = 1
    healthy: bool = True
    active_connections: int = 0
```

```
class LoadBalancer:
```

```
    def __init__(self):
        self.instances: List[ServerInstance] = []
        self.health_check_interval = 10
```

```
    def add_instance(self, instance: ServerInstance):
        self.instances.append(instance)
```

```
    def remove_instance(self, instance_id: str):
        self.instances = [i for i in self.instances if i.id != instance_id]
```

```
def get_healthy_instances(self) -> List[ServerInstance]:  
    return [i for i in self.instances if i.healthy]
```

```
def select_instance_round_robin(self) -> Optional[ServerInstance]:  
    healthy = self.get_healthy_instances()  
    if not healthy:  
        return None
```

```
    selected = healthy[0]  
    self.instances.remove(selected)  
    self.instances.append(selected)  
    return selected
```

```
def select_instance_least_connections(self) ->  
Optional[ServerInstance]:  
    healthy = self.get_healthy_instances()  
    if not healthy:  
        return None
```

```
    return min(healthy, key=lambda i: i.active_connections)
```

```
def select_instance_weighted_random(self) ->  
Optional[ServerInstance]:  
    healthy = self.get_healthy_instances()  
    if not healthy:  
        return None
```

```
    total_weight = sum(i.weight for i in healthy)  
    rand = random.uniform(0, total_weight)
```

```
    cumulative = 0  
    for instance in healthy:  
        cumulative += instance.weight  
        if rand <= cumulative:  
            return instance
```

```
    return healthy[-1]
```

```
async def health_check_loop(self, check_func):  
    while True:
```



```

for instance in self.instances:
    try:
        healthy = await check_func(instance.host, instance.port)
        instance.healthy = healthy
    except Exception:
        instance.healthy = False

await asyncio.sleep(self.health_check_interval)

```

Database Replication

```
class DatabaseCluster:
```

```

    def __init__(self):
        self.primary: Optional[ServerInstance] = None
        self.replicas: List[ServerInstance] = []
        self.read_preference = "replica_preferred"

```

```

    def set_primary(self, instance: ServerInstance):
        self.primary = instance

```

```

    def add_replica(self, instance: ServerInstance):
        self.replicas.append(instance)

```

```

    def get_write_connection(self) -> Optional[ServerInstance]:
        if self.primary and self.primary.healthy:
            return self.primary
        return None

```

```

    def get_read_connection(self) -> Optional[ServerInstance]:
        if self.read_preference == "primary_only":
            return self.primary if self.primary and self.primary.healthy else
            None

```

```

    healthy_replicas = [r for r in self.replicas if r.healthy]

```

```

    if self.read_preference == "replica_only":
        if healthy_replicas:
            return min(healthy_replicas, key=lambda r: r.active_connections)

```

```
return None
```

```
if self.read_preference == "replica_preferred":
```

```
if healthy_replicas:
```

```
return min(healthy_replicas, key=lambda r: r.active_connections)
```

```
return self.primary if self.primary and self.primary.healthy else
```

```
None
```

```
if self.read_preference == "primary_preferred":
```

```
if self.primary and self.primary.healthy:
```

```
return self.primary
```

```
if healthy_replicas:
```

```
return min(healthy_replicas, key=lambda r: r.active_connections)
```

```
return None
```

```
return None
```

```
async def promote_replica(self, replica_id: str) -> bool:
```

```
replica = next((r for r in self.replicas if r.id == replica_id), None)
```

```
if not replica:
```

```
return False
```

```
old_primary = self.primary
```

```
self.primary = replica
```

```
self.replicas.remove(replica)
```

```
if old_primary:
```

```
self.replicas.append(old_primary)
```

```
return True
```

SECTION 3: FAILOVER

Automatic Failover

```
from enum import Enum
```

```
from typing import Callable, Awaitable
```

```
import time
```

```
class FailoverState(Enum):
    NORMAL = "normal"
    DETECTING = "detecting"
    FAILING_OVER = "failing_over"
    FAILED_OVER = "failed_over"
```

```
class FailoverManager:
```

```
    def __init__(
        self,
        primary_check: Callable[[], Awaitable[bool]],
        secondary_check: Callable[[], Awaitable[bool]],
        switch_to_secondary: Callable[[], Awaitable[bool]],
        switch_to_primary: Callable[[], Awaitable[bool]]
    ):
        self.primary_check = primary_check
        self.secondary_check = secondary_check
        self.switch_to_secondary = switch_to_secondary
        self.switch_to_primary = switch_to_primary
```

```
        self.state = FailoverState.NORMAL
        self.failure_count = 0
        self.failure_threshold = 3
        self.check_interval = 5
        self.last_failover_time = 0
        self.min_failover_interval = 300
```

```
    async def monitor_loop(self):
        while True:
            await self._check_and_failover()
            await asyncio.sleep(self.check_interval)
```

```
    async def _check_and_failover(self):
        if self.state == FailoverState.NORMAL:
            primary_healthy = await self._safe_check(self.primary_check)
```

```
            if not primary_healthy:
                self.failure_count += 1
```

```
if self.failure_count >= self.failure_threshold:
```

```
    await self._initiate_failover()
```

```
else:
```

```
    self.failure_count = 0
```

```
elif self.state == FailoverState.FAILED_OVER:
```

```
    primary_healthy = await self._safe_check(self.primary_check)
```

```
if primary_healthy:
```

```
    time_since_failover = time.time() - self.last_failover_time
```

```
    if time_since_failover >= self.min_failover_interval:
```

```
        await self._initiate_failback()
```

```
async def _safe_check(self, check: Callable) -> bool:
```

```
    try:
```

```
        return await asyncio.wait_for(check(), timeout=5.0)
```

```
    except Exception:
```

```
        return False
```

```
async def _initiate_failover(self):
```

```
    self.state = FailoverState.FAILING_OVER
```

```
    secondary_healthy = await self._safe_check(self.secondary_check)
```

```
    if not secondary_healthy:
```

```
        self.state = FailoverState.NORMAL
```

```
    return
```

```
    success = await self.switch_to_secondary()
```

```
    if success:
```

```
        self.state = FailoverState.FAILED_OVER
```

```
        self.last_failover_time = time.time()
```

```
        self.failure_count = 0
```

```
    else:
```

```
        self.state = FailoverState.NORMAL
```

```
async def _initiate_failback(self):
```

```
    self.state = FailoverState.FAILING_OVER
```

```
    success = await self.switch_to_primary()
```

```
    if success:
```

```
self.state = FailoverState.NORMAL
self.failure_count = 0
else:
self.state = FailoverState.FAILED_OVER
```

Circuit Breaker Pattern

```
class CircuitState(Enum):
    CLOSED = "closed"
    OPEN = "open"
    HALF_OPEN = "half_open"
```

```
class CircuitBreaker:
```

```
    def __init__(
        self,
        failure_threshold: int = 5,
        recovery_timeout: float = 30.0,
        half_open_max_calls: int = 3
    ):
        self.failure_threshold = failure_threshold
        self.recovery_timeout = recovery_timeout
        self.half_open_max_calls = half_open_max_calls
```

```
        self.state = CircuitState.CLOSED
        self.failure_count = 0
        self.last_failure_time = 0
        self.half_open_calls = 0
```

```
    async def call(self, func: Callable, *args, **kwargs):
        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time >= self.recovery_timeout:
                self.state = CircuitState.HALF_OPEN
                self.half_open_calls = 0
            else:
                raise CircuitOpenError("Circuit is open")
```

```
        if self.state == CircuitState.HALF_OPEN:
```

```
if self.half_open_calls >= self.half_open_max_calls:
    raise CircuitOpenError("Circuit is half-open, max calls reached")
    self.half_open_calls += 1
```

```
try:
    result = await func(*args, **kwargs)
    self._on_success()
    return result
except Exception as e:
    self._on_failure()
    raise
```

```
def _on_success(self):
    if self.state == CircuitState.HALF_OPEN:
        self.state = CircuitState.CLOSED
        self.failure_count = 0
```

```
def _on_failure(self):
    self.failure_count += 1
    self.last_failure_time = time.time()
```

```
if self.failure_count >= self.failure_threshold:
    self.state = CircuitState.OPEN
```

```
def reset(self):
    self.state = CircuitState.CLOSED
    self.failure_count = 0
```

```
class CircuitOpenError(Exception):
    pass
```

SECTION 4: MULTI-REGION DEPLOYMENT

Geographic Distribution

```
class MultiRegionDeployment:
```

```
    def __init__(self):
```

```
self.regions: Dict[str, Dict] = {}
self.primary_region: Optional[str] = None
```

```
def add_region(
    self,
    region_id: str,
    endpoints: List[str],
    is_primary: bool = False
):
    self.regions[region_id] = {
        "endpoints": endpoints,
        "healthy": True,
        "latency_ms": 0
    }
```

```
if is_primary:
    self.primary_region = region_id
```

```
def get_nearest_region(self, user_region: str) -> Optional[str]:
    region_distances = {
        ("us-east", "us-west"): 70,
        ("us-east", "eu-west"): 80,
        ("us-east", "ap-southeast"): 200,
        ("us-west", "eu-west"): 140,
        ("us-west", "ap-southeast"): 120,
        ("eu-west", "ap-southeast"): 180
    }
```

```
healthy_regions = [r for r, data in self.regions.items() if
data["healthy"]]
```

```
if user_region in healthy_regions:
    return user_region
```

```
min_distance = float("inf")
nearest = None
```

```
for region in healthy_regions:
    key = tuple(sorted([user_region, region]))
    distance = region_distances.get(key, 100)
```

```
if distance < min_distance:
    min_distance = distance
    nearest = region
```

```
return nearest or self.primary_region
```

```
def get_failover_region(self, failed_region: str) -> Optional[str]:
    healthy = [r for r in self.regions if r != failed_region and
self.regions[r]["healthy"]]
```

```
if not healthy:
    return None
```

```
if self.primary_region in healthy and failed_region !=
self.primary_region:
    return self.primary_region
```

```
return healthy[0]
```

Global Load Balancing DNS

```
class GeoDNSConfig:
```

```
    def __init__(self, domain: str):
        self.domain = domain
        self.records: Dict[str, List[Dict]] = {}
        self.health_checks: List[Dict] = []
```

```
    def add_record(
        self,
        subdomain: str,
        record_type: str,
        values: List[str],
        ttl: int = 60,
        geo_location: Optional[str] = None,
        weight: int = 100
    ):
        key = subdomain or "@"
```



```

if key not in self.records:
    self.records[key] = []

self.records[key].append({
    "type": record_type,
    "values": values,
    "ttl": ttl,
    "geo_location": geo_location,
    "weight": weight
})

```

```

def add_health_check(
    self,
    endpoint: str,
    protocol: str = "HTTPS",
    port: int = 443,
    path: str = "/health",
    interval: int = 30,
    threshold: int = 3
):
    self.health_checks.append({
        "endpoint": endpoint,
        "protocol": protocol,
        "port": port,
        "path": path,
        "interval": interval,
        "threshold": threshold
    })

```

```

def to_route53_config(self) -> Dict:
    return {
        "hosted_zone": self.domain,
        "records": self.records,
        "health_checks": self.health_checks
    }

```

SECTION 5: DATA CONSISTENCY

Eventual Consistency Handling

```

class EventualConsistencyManager:

    def __init__(self, max_lag_seconds: float = 5.0):
        self.max_lag = max_lag_seconds
        self.write_timestamps: Dict[str, float] = {}

    def record_write(self, key: str):
        self.write_timestamps[key] = time.time()

    def should_read_from_primary(self, key: str) -> bool:
        if key not in self.write_timestamps:
            return False

        age = time.time() - self.write_timestamps[key]
        return age < self.max_lag

    def get_read_preference(self, key: str) -> str:
        if self.should_read_from_primary(key):
            return "primary"
        return "replica"

    def cleanup_old_entries(self, max_age: float = 60.0):
        now = time.time()
        self.write_timestamps = {
            k: v for k, v in self.write_timestamps.items()
            if now - v < max_age
        }

```

Distributed Locks

```

import redis
import uuid

class DistributedLock:

    def __init__(self, redis_client: redis.Redis, lock_name: str, ttl: int =
30):

```

```
self.redis = redis_client
self.lock_name = f"lock:{lock_name}"
self.ttl = ttl
self.token = str(uuid.uuid4())
self.acquired = False
```

```
async def acquire(self, timeout: float = 10.0) -> bool:
    start = time.time()
```

```
    while time.time() - start < timeout:
        acquired = await self.redis.set(
            self.lock_name,
            self.token,
            nx=True,
            ex=self.ttl
        )
```

```
        if acquired:
            self.acquired = True
            return True
```

```
    await asyncio.sleep(0.1)
```

```
    return False
```

```
async def release(self) -> bool:
    if not self.acquired:
        return False
```

```
    script = """
    if redis.call("get", KEYS[1]) == ARGV[1] then
        return redis.call("del", KEYS[1])
    else
        return 0
    end
    """
```

```
    result = await self.redis.eval(script, 1, self.lock_name, self.token)
    self.acquired = False
    return result == 1
```

```

async def extend(self, additional_ttl: int) -> bool:
    if not self.acquired:
        return False

```

```

    script = """
    if redis.call("get", KEYS[1]) == ARGV[1] then
        return redis.call("expire", KEYS[1], ARGV[2])
    else
        return 0
    end
    """

```

```

    result = await self.redis.eval(
        script, 1, self.lock_name, self.token, self.ttl + additional_ttl
    )
    return result == 1

```

```

async def _aenter_(self):
    acquired = await self.acquire()
    if not acquired:
        raise LockAcquisitionError(f'Could not acquire lock {self.lock_name}')
    return self

```

```

async def _aexit_(self, exc_type, exc_val, exc_tb):
    await self.release()

```

```

class LockAcquisitionError(Exception):
    pass

```

SECTION 6: GRACEFUL DEGRADATION

Service Degradation

```

class ServiceDegradation:

    def __init__(self):

```

```

self.degradation_levels: Dict[str, int] = {}
self.feature_dependencies: Dict[str, List[str]] = {}

def set_degradation_level(self, service: str, level: int):
    self.degradation_levels[service] = level

def register_feature(self, feature: str, required_services: List[str]):
    self.feature_dependencies[feature] = required_services

def is_feature_available(self, feature: str) -> bool:
    if feature not in self.feature_dependencies:
        return True

    required = self.feature_dependencies[feature]
    for service in required:
        if self.degradation_levels.get(service, 0) >= 2:
            return False

    return True

def get_available_features(self) -> List[str]:
    return [f for f in self.feature_dependencies if
self.is_feature_available(f)]

```

```

class GracefulDegradation:

```

```

    def __init__(self):
        self.fallbacks: Dict[str, Callable] = {}
        self.cache: Dict[str, tuple] = {}

    def register_fallback(self, operation: str, fallback: Callable):
        self.fallbacks[operation] = fallback

    def cache_result(self, operation: str, result, ttl: float = 300):
        self.cache[operation] = (result, time.time() + ttl)

    def get_cached_result(self, operation: str):
        if operation in self.cache:
            result, expiry = self.cache[operation]

```

```
if time.time() < expiry:
    return result
return None
```

```
async def execute_with_fallback(
    self,
    operation: str,
    primary_func: Callable,
    *args,
    **kwargs
):
    try:
        result = await primary_func(*args, **kwargs)
        self.cache_result(operation, result)
        return result
    except Exception as e:
        cached = self.get_cached_result(operation)
        if cached is not None:
            return cached

    if operation in self.fallbacks:
        return await self.fallbacks[operation](*args, **kwargs)

    raise
```

SECTION 7: ZERO-DOWNTIME DEPLOYMENTS

Rolling Updates

```
class RollingUpdateManager:

    def __init__(
        self,
        instances: List[ServerInstance],
        max_unavailable: int = 1,
        health_check_timeout: float = 60.0
    ):
        self.instances = instances
        self.max_unavailable = max_unavailable
```

```
self.health_check_timeout = health_check_timeout
```

```
async def perform_rolling_update(
    self,
    deploy_func: Callable[[ServerInstance], Awaitable[bool]],
    health_check: Callable[[ServerInstance], Awaitable[bool]]
) -> Dict[str, List[str]]:
```

```
    results = {"success": [], "failed": []}
```

```
    for i in range(0, len(self.instances), self.max_unavailable):
        batch = self.instances[i:i + self.max_unavailable]
```

```
        for instance in batch:
            instance.healthy = False
```

```
        await asyncio.sleep(5)
```

```
        for instance in batch:
            try:
                success = await deploy_func(instance)
            if not success:
                results["failed"].append(instance.id)
            continue
```

```
        healthy = await self._wait_for_health(instance, health_check)
        if healthy:
            instance.healthy = True
            results["success"].append(instance.id)
        else:
            results["failed"].append(instance.id)
```

```
    except Exception as e:
        results["failed"].append(instance.id)
```

```
    if results["failed"]:
        break
```

```
    return results
```

```
async def _wait_for_health(  
    self,  
    instance: ServerInstance,  
    health_check: Callable  
) -> bool:
```

```
    start = time.time()
```

```
    while time.time() - start < self.health_check_timeout:
```

```
        try:
```

```
            if await health_check(instance):
```

```
                return True
```

```
        except Exception:
```

```
            pass
```

```
        await asyncio.sleep(2)
```

```
    return False
```

Canary Deployments

```
class CanaryDeployment:
```

```
    def __init__(
```

```
        self,
```

```
        canary_percentage: float = 5.0,
```

```
        promotion_interval: float = 300.0,
```

```
        max_error_rate: float = 0.01
```

```
    ):
```

```
        self.canary_percentage = canary_percentage
```

```
        self.promotion_interval = promotion_interval
```

```
        self.max_error_rate = max_error_rate
```

```
        self.current_percentage = 0.0
```

```
        self.canary_active = False
```

```
    def should_route_to_canary(self) -> bool:
```

```
        if not self.canary_active:
```

```
            return False
```



```

import random
return random.random() * 100 < self.current_percentage

async def start_canary(
self,
deploy_canary: Callable[[], Awaitable[bool]],
get_error_rate: Callable[[str], Awaitable[float]],
promote_full: Callable[[], Awaitable[bool]],
rollback: Callable[[], Awaitable[bool]]
) -> bool:

    success = await deploy_canary()
    if not success:
        return False

    self.canary_active = True
    self.current_percentage = self.canary_percentage

    while self.current_percentage < 100:
        await asyncio.sleep(self.promotion_interval)

        error_rate = await get_error_rate("canary")
        baseline_error_rate = await get_error_rate("stable")

        if error_rate > self.max_error_rate or error_rate >
baseline_error_rate * 2:
            await rollback()
            self.canary_active = False
            return False

        self.current_percentage = min(self.current_percentage * 2, 100)

    success = await promote_full()
    self.canary_active = False
    return success

```

This chapter covered high availability patterns for crypto and Telegram applications. The next chapter covers scaling strategies for handling growth.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 4: Scaling Strategies

Success brings traffic. Traffic demands scale. This chapter teaches you to scale crypto and Telegram applications from hundreds to millions of users without breaking.

SECTION 1: SCALING FUNDAMENTALS

Vertical vs Horizontal Scaling

Vertical Scaling (Scale Up):

Add more resources to existing servers. More CPU. More RAM. Faster disks. Simple but limited. Eventually you hit hardware limits.

Horizontal Scaling (Scale Out):

Add more servers. Distribute load across machines. More complex but unlimited. The path for serious growth.

Bottleneck Identification

Before scaling, find what limits you:

```
from dataclasses import dataclass
from typing import Dict, List
import asyncio
```

```
@dataclass
class ResourceMetrics:
    cpu_percent: float
    memory_percent: float
    disk_io_percent: float
    network_io_percent: float
    database_connections: int
    database_query_time_ms: float
    cache_hit_rate: float
    request_queue_length: int
```

```
class BottleneckAnalyzer:
```

```
    def __init__(self):
        self.thresholds = {
            "cpu_percent": 80,
            "memory_percent": 85,
            "disk_io_percent": 70,
            "network_io_percent": 70,
            "database_connections": 80,
            "database_query_time_ms": 100,
            "cache_hit_rate": 0.90,
            "request_queue_length": 100
        }
```

```
    def analyze(self, metrics: ResourceMetrics) -> List[Dict]:
        bottlenecks = []
```

```
        if metrics.cpu_percent > self.thresholds["cpu_percent"]:
            bottlenecks.append({
                "resource": "cpu",
                "severity": "high" if metrics.cpu_percent > 90 else "medium",
                "recommendation": "Add more CPU cores or optimize code"
            })
```

```
        if metrics.memory_percent > self.thresholds["memory_percent"]:
            bottlenecks.append({
                "resource": "memory",
                "severity": "high" if metrics.memory_percent > 95 else "medium",
                "recommendation": "Add more RAM or fix memory leaks"
            })
```

```
        if metrics.database_query_time_ms >
            self.thresholds["database_query_time_ms"]:
            bottlenecks.append({
                "resource": "database",
                "severity": "high",
                "recommendation": "Add read replicas or optimize queries"
            })
```

```

if metrics.cache_hit_rate < self.thresholds["cache_hit_rate"]:
    bottlenecks.append({
        "resource": "cache",
        "severity": "medium",
        "recommendation": "Increase cache size or review cache strategy"
    })

return bottlenecks

```

SECTION 2: APPLICATION SCALING

Stateless Design

Stateless applications scale easily:

```
class StatelessApplicationDesign:
```

```

    def __init__(self):
        self.session_store = None
        self.cache = None

```

```

    def configure_external_state(
        self,
        session_store_url: str,
        cache_url: str
    ):
        import redis

```

```

        self.session_store = redis.from_url(session_store_url)
        self.cache = redis.from_url(cache_url)

```

```

    async def get_session(self, session_id: str) -> Dict:
        import json

```

```

        data = await self.session_store.get(f'session:{session_id}')
        if data:
            return json.loads(data)
        return {}

```

```

async def set_session(self, session_id: str, data: Dict, ttl: int =
3600):
    import json

    await self.session_store.setex(
        f'session:{session_id}',
        ttl,
        json.dumps(data)
    )

    async def cache_get(self, key: str):
        import json

        data = await self.cache.get(key)
        if data:
            return json.loads(data)
        return None

    async def cache_set(self, key: str, value, ttl: int = 300):
        import json

        await self.cache.setex(key, ttl, json.dumps(value))

```

Worker Pools

```

from concurrent.futures import ProcessPoolExecutor,
ThreadPoolExecutor
import multiprocessing

class WorkerPoolManager:

    def __init__(self):
        cpu_count = multiprocessing.cpu_count()

        self.cpu_bound_pool =
        ProcessPoolExecutor(max_workers=cpu_count)
        self.io_bound_pool =
        ThreadPoolExecutor(max_workers=cpu_count * 4)

```

```

async def run_cpu_bound(self, func, *args):
    loop = asyncio.get_event_loop()
    return await loop.run_in_executor(self.cpu_bound_pool, func, *args)

async def run_io_bound(self, func, *args):
    loop = asyncio.get_event_loop()
    return await loop.run_in_executor(self.io_bound_pool, func, *args)

def shutdown(self):
    self.cpu_bound_pool.shutdown(wait=True)
    self.io_bound_pool.shutdown(wait=True)

```

SECTION 3: DATABASE SCALING

Read Replicas

```

class ReadReplicaRouter:

    def __init__(self, primary_url: str, replica_urls: List[str]):
        self.primary_url = primary_url
        self.replica_urls = replica_urls
        self.current_replica_index = 0

    def get_write_url(self) -> str:
        return self.primary_url

    def get_read_url(self) -> str:
        if not self.replica_urls:
            return self.primary_url

        url = self.replica_urls[self.current_replica_index]
        self.current_replica_index = (self.current_replica_index + 1) %
len(self.replica_urls)
        return url

```

Connection Pooling

```
from sqlalchemy import create_engine
from sqlalchemy.pool import QueuePool
```

```
class DatabaseConnectionManager:
```

```
    def __init__(
        self,
        primary_url: str,
        replica_urls: List[str],
        pool_size: int = 20,
        max_overflow: int = 10
    ):
        self.primary_engine = self._create_engine(primary_url, pool_size,
max_overflow)
```

```
        self.replica_engines = [
            self._create_engine(url, pool_size // 2, max_overflow // 2)
            for url in replica_urls
        ]
```

```
        self.replica_index = 0
```

```
    def _create_engine(self, url: str, pool_size: int, max_overflow: int):
        return create_engine(
            url,
            poolclass=QueuePool,
            pool_size=pool_size,
            max_overflow=max_overflow,
            pool_pre_ping=True,
            pool_recycle=3600
        )
```

```
    def get_write_engine(self):
        return self.primary_engine
```

```
    def get_read_engine(self):
        if not self.replica_engines:
            return self.primary_engine
```

```
engine = self.replica_engines[self.replica_index]
self.replica_index = (self.replica_index + 1) %
len(self.replica_engines)
return engine
```

Database Sharding

```
class ShardingStrategy:
```

```
    def __init__(self, shard_count: int, shard_urls: Dict[int, str]):
        self.shard_count = shard_count
        self.shard_urls = shard_urls
```

```
    def get_shard_for_key(self, key: str) -> int:
        hash_value = hash(key)
        return hash_value % self.shard_count
```

```
    def get_shard_url(self, shard_id: int) -> str:
        return self.shard_urls[shard_id]
```

```
    def route_query(self, user_id: str) -> str:
        shard_id = self.get_shard_for_key(user_id)
        return self.get_shard_url(shard_id)
```

```
class ConsistentHashSharding:
```

```
    def __init__(self, nodes: List[str], virtual_nodes: int = 150):
        self.ring: Dict[int, str] = {}
        self.sorted_keys: List[int] = []
```

```
    for node in nodes:
        for i in range(virtual_nodes):
            key = self._hash(f'{node}:{i}')
            self.ring[key] = node
```

```
    self.sorted_keys = sorted(self.ring.keys())
```

```
    def _hash(self, key: str) -> int:
```



```

import hashlib
return int(hashlib.md5(key.encode()).hexdigest(), 16)

def get_node(self, key: str) -> str:
    if not self.ring:
        return ""

    hash_key = self._hash(key)

    for ring_key in self.sorted_keys:
        if hash_key <= ring_key:
            return self.ring[ring_key]

    return self.ring[self.sorted_keys[0]]

```

SECTION 4: CACHING STRATEGIES

Multi-Level Caching

```

from typing import Optional, Any
import time

class LocalCache:

    def __init__(self, max_size: int = 1000):
        self.cache: Dict[str, tuple] = {}
        self.max_size = max_size

    def get(self, key: str) -> Optional[Any]:
        if key in self.cache:
            value, expiry = self.cache[key]
            if time.time() < expiry:
                return value
            del self.cache[key]
        return None

    def set(self, key: str, value: Any, ttl: int = 60):
        if len(self.cache) >= self.max_size:

```

```
oldest_key = min(self.cache, key=lambda k: self.cache[k][1])
del self.cache[oldest_key]
```

```
self.cache[key] = (value, time.time() + ttl)
```

```
class MultiLevelCache:
```

```
def __init__(self, redis_client):
    self.l1_cache = LocalCache(max_size=10000)
    self.l2_cache = redis_client
```

```
async def get(self, key: str) -> Optional[Any]:
    value = self.l1_cache.get(key)
    if value is not None:
        return value
```

```
import json
data = await self.l2_cache.get(key)
if data:
    value = json.loads(data)
    self.l1_cache.set(key, value, ttl=30)
    return value
```

```
return None
```

```
async def set(self, key: str, value: Any, ttl: int = 300):
    import json
```

```
    self.l1_cache.set(key, value, ttl=min(ttl, 60))
    await self.l2_cache.setex(key, ttl, json.dumps(value))
```

```
async def invalidate(self, key: str):
    if key in self.l1_cache.cache:
        del self.l1_cache.cache[key]
    await self.l2_cache.delete(key)
```

Cache-Aside Pattern

```
class CacheAsideRepository:
```

```
    def __init__(self, cache: MultiLevelCache, database):  
        self.cache = cache  
        self.db = database
```

```
    async def get_user(self, user_id: str) -> Optional[Dict]:  
        cache_key = f"user:{user_id}"
```

```
        cached = await self.cache.get(cache_key)  
        if cached:  
            return cached
```

```
        user = await self.db.fetch_one(  
            "SELECT * FROM users WHERE id = :id",  
            {"id": user_id}  
        )
```

```
        if user:  
            user_dict = dict(user)  
            await self.cache.set(cache_key, user_dict, ttl=300)  
            return user_dict
```

```
        return None
```

```
    async def update_user(self, user_id: str, data: Dict):  
        await self.db.execute(  
            "UPDATE users SET data = :data WHERE id = :id",  
            {"id": user_id, "data": data}  
        )
```

```
        cache_key = f"user:{user_id}"  
        await self.cache.invalidate(cache_key)
```

SECTION 5: MESSAGE QUEUE SCALING

Queue-Based Architecture

```
import json
```

```
from typing import Callable, Awaitable
```

```
class MessageQueue:
```

```
    def __init__(self, redis_client):  
        self.redis = redis_client
```

```
    async def publish(self, queue_name: str, message: Dict):  
        await self.redis.lpush(queue_name, json.dumps(message))
```

```
    async def consume(  
        self,  
        queue_name: str,  
        handler: Callable[[Dict], Awaitable[None]],  
        batch_size: int = 10  
    ):  
        while True:  
            messages = await self.redis.lrange(queue_name, -batch_size, -1)
```

```
            if not messages:  
                await asyncio.sleep(0.1)  
                continue
```

```
            for msg_data in messages:  
                message = json.loads(msg_data)  
                try:  
                    await handler(message)  
                    await self.redis.rpop(queue_name)  
                except Exception as e:  
                    await self._handle_failed_message(queue_name, message, e)
```

```
    async def _handle_failed_message(self, queue_name: str, message:  
Dict, error: Exception):  
        message["_error"] = str(error)  
        message["_retry_count"] = message.get("_retry_count", 0) + 1
```

```
        if message["_retry_count"] < 3:  
            await self.redis.lpush(f'{queue_name}:retry', json.dumps(message))  
        else:
```

```
await self.redis.lpush(f'{queue_name}:dead', json.dumps(message))
```

```
class WorkerScaler:
```

```
    def __init__(self, queue: MessageQueue):
```

```
        self.queue = queue
```

```
        self.workers: Dict[str, List[asyncio.Task]] = {}
```

```
    async def scale_workers(
```

```
        self,
```

```
        queue_name: str,
```

```
        handler: Callable,
```

```
        min_workers: int = 1,
```

```
        max_workers: int = 10
```

```
    ):
```

```
        while True:
```

```
            queue_length = await self.queue.redis.llen(queue_name)
```

```
            desired_workers = min(max(queue_length // 100, min_workers),  
max_workers)
```

```
            current_workers = len(self.workers.get(queue_name, []))
```

```
            if desired_workers > current_workers:
```

```
                for _ in range(desired_workers - current_workers):
```

```
                    task = asyncio.create_task(
```

```
                        self.queue.consume(queue_name, handler)
```

```
                    )
```

```
                if queue_name not in self.workers:
```

```
                    self.workers[queue_name] = []
```

```
                    self.workers[queue_name].append(task)
```

```
            elif desired_workers < current_workers:
```

```
                workers = self.workers[queue_name]
```

```
                for _ in range(current_workers - desired_workers):
```

```
                    task = workers.pop()
```

```
                    task.cancel()
```

```
            await asyncio.sleep(30)
```

SECTION 6: AUTO-SCALING

Kubernetes Horizontal Pod Autoscaler

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: crypto-bot-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: crypto-bot
  minReplicas: 3
  maxReplicas: 20
  metrics:
    - type: Resource
      resource:
        name: cpu
      target:
        type: Utilization
        averageUtilization: 70
    - type: Resource
      resource:
        name: memory
      target:
        type: Utilization
        averageUtilization: 80
    - type: Pods
      pods:
        metric:
          name: requests_per_second
        target:
          type: AverageValue
          averageValue: "100"
      behavior:
        scaleUp:
          stabilizationWindowSeconds: 60
  policies:
```

- type: Pods
value: 4
periodSeconds: 60
scaleDown:
stabilizationWindowSeconds: 300
policies:
 - type: Percent
value: 25
periodSeconds: 120

Custom Auto-Scaling Logic

class CustomAutoScaler:

```
def __init__(
    self,
    min_instances: int = 2,
    max_instances: int = 20,
    scale_up_threshold: float = 0.7,
    scale_down_threshold: float = 0.3
):
    self.min_instances = min_instances
    self.max_instances = max_instances
    self.scale_up_threshold = scale_up_threshold
    self.scale_down_threshold = scale_down_threshold
    self.current_instances = min_instances
```

```
async def evaluate_scaling(
    self,
    metrics: Dict[str, float]
) -> Dict[str, Any]:
```

```
cpu_utilization = metrics.get("cpu_utilization", 0)
memory_utilization = metrics.get("memory_utilization", 0)
request_rate = metrics.get("request_rate", 0)
request_latency = metrics.get("request_latency_p95", 0)
```

```
scale_score = 0
```

```
if cpu_utilization > self.scale_up_threshold:
```

```

scale_score += 1
elif cpu_utilization < self.scale_down_threshold:
scale_score -= 1

if memory_utilization > self.scale_up_threshold:
scale_score += 1
elif memory_utilization < self.scale_down_threshold:
scale_score -= 1

if request_latency > 1000:
scale_score += 2

action = "none"
new_instances = self.current_instances

if scale_score >= 2:
new_instances = min(self.current_instances + 2,
self.max_instances)
action = "scale_up"
elif scale_score <= -2:
new_instances = max(self.current_instances - 1, self.min_instances)
action = "scale_down"

return {
"action": action,
"current_instances": self.current_instances,
"target_instances": new_instances,
"scale_score": scale_score
}

```

SECTION 7: BLOCKCHAIN NODE SCALING

Multi-Node Configuration

```

class BlockchainNodePool:

    def __init__(self):
self.nodes: List[Dict] = []
self.current_index = 0

```



```
def add_node(self, url: str, priority: int = 1, rate_limit: int = 100):
    self.nodes.append({
        "url": url,
        "priority": priority,
        "rate_limit": rate_limit,
        "requests_this_second": 0,
        "last_reset": time.time(),
        "healthy": True,
        "latency_ms": 0
    })
```

```
def get_best_node(self) -> Optional[str]:
    healthy_nodes = [n for n in self.nodes if n["healthy"]]
```

```
    if not healthy_nodes:
        return None
```

```
    for node in healthy_nodes:
        now = time.time()
        if now - node["last_reset"] >= 1:
            node["requests_this_second"] = 0
            node["last_reset"] = now
```

```
    available = [
        n for n in healthy_nodes
        if n["requests_this_second"] < n["rate_limit"]
    ]
```

```
    if not available:
        return healthy_nodes[0]["url"]
```

```
    best = min(available, key=lambda n: (-n["priority"],
        n["latency_ms"]))
    best["requests_this_second"] += 1
    return best["url"]
```

```
async def health_check_all(self, web3_class):
    for node in self.nodes:
        try:
```

```

w3 = web3_class(web3_class.HTTPProvider(node["url"]))
start = time.time()
await w3.eth.block_number
node["latency_ms"] = (time.time() - start) * 1000
node["healthy"] = True
except Exception:
node["healthy"] = False

```

This chapter covered scaling strategies for handling growth. The next chapter covers security hardening for production systems.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 5: Security Hardening

Production systems are targets. Every exposed endpoint is an attack surface. This chapter teaches you to harden your crypto and Telegram infrastructure against real-world attacks.

SECTION 1: NETWORK SECURITY

Firewall Configuration

```
class FirewallRules:
```

```
    @staticmethod
```

```
    def generate_iptables_rules() -> str:
```

```
        return """
```

```
*filter
```

```
:INPUT DROP [0:0]
```

```
:FORWARD DROP [0:0]
```

```
:OUTPUT ACCEPT [0:0]
```

```
-A INPUT -i lo -j ACCEPT
```

```
-A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
```

```
-A INPUT -p tcp --dport 22 -s 10.0.0.0/8 -j ACCEPT
```

```
-A INPUT -p tcp --dport 443 -j ACCEPT
```

```
-A INPUT -p tcp --dport 80 -j ACCEPT
-A INPUT -s 10.0.0.0/8 -p tcp --dport 8000 -j ACCEPT
-A INPUT -p icmp --icmp-type echo-request -j ACCEPT
-A INPUT -j LOG --log-prefix "IPTables-Dropped: "
-A INPUT -j DROP
```

COMMIT

"""

@staticmethod

def generate_security_group_rules() -> List[Dict]:

return [

{

"direction": "ingress",

"protocol": "tcp",

"port": 443,

"source": "0.0.0.0/0",

"description": "HTTPS from anywhere"

},

{

"direction": "ingress",

"protocol": "tcp",

"port": 22,

"source": "10.0.0.0/8",

"description": "SSH from internal only"

},

{

"direction": "ingress",

"protocol": "tcp",

"port": 5432,

"source": "10.0.1.0/24",

"description": "PostgreSQL from app subnet"

},

{

"direction": "ingress",

"protocol": "tcp",

"port": 6379,

```
"source": "10.0.1.0/24",
"description": "Redis from app subnet"
}
]
```

VPN Configuration

```
class WireGuardConfig:
```

```
    @staticmethod
    def generate_server_config(
        private_key: str,
        address: str,
        listen_port: int,
        peers: List[Dict]
    ) -> str:

        config = f"""
[Interface]
PrivateKey = {private_key}
Address = {address}
ListenPort = {listen_port}
PostUp = iptables -A FORWARD -i wg0 -j ACCEPT
PostDown = iptables -D FORWARD -i wg0 -j ACCEPT

"""

        for peer in peers:
            config += f"""
[Peer]
PublicKey = {peer['public_key']}
AllowedIPs = {peer['allowed_ips']}
"""

        return config
```

SECTION 2: APPLICATION SECURITY

Input Sanitization

```
import re
from typing import Any, Optional
```

```
class InputSanitizer:
```

```
    @staticmethod
```

```
    def sanitize_string(value: str, max_length: int = 1000) -> str:
        if not isinstance(value, str):
            return ""
```

```
        value = value[:max_length]
```

```
        value = re.sub(r'[\x00-\x08\x0b\x0c\x0e-\x1f\x7f-\x9f]', '', value)
        return value
```

```
    @staticmethod
```

```
    def sanitize_ethereum_address(address: str) -> Optional[str]:
        if not isinstance(address, str):
            return None
```

```
        address = address.strip().lower()
```

```
        if not re.match(r'^0x[a-f0-9]{40}$', address):
            return None
```

```
        return address
```

```
    @staticmethod
```

```
    def sanitize_amount(value: Any, max_value: int = 10**30) ->
Optional[int]:
```

```
        try:
```

```
            amount = int(value)
```

```
            if amount < 0 or amount > max_value:
```

```
                return None
```

```
            return amount
```

```
        except (ValueError, TypeError):
```

```
            return None
```

```
    @staticmethod
```

```
    def sanitize_telegram_user_id(user_id: Any) -> Optional[int]:
        try:
```

```
uid = int(user_id)
if uid <= 0 or uid > 10**15:
    return None
return uid
except (ValueError, TypeError):
    return None
```

Rate Limiting

```
import time
from collections import defaultdict
```

```
class RateLimiter:

    def __init__(self):
        self.requests: Dict[str, List[float]] = defaultdict(list)

    def is_allowed(
        self,
        key: str,
        max_requests: int,
        window_seconds: int
    ) -> bool:

        now = time.time()
        window_start = now - window_seconds

        self.requests[key] = [
            t for t in self.requests[key]
            if t > window_start
        ]

        if len(self.requests[key]) >= max_requests:
            return False

        self.requests[key].append(now)
        return True
```

```
def get_remaining(  
    self,  
    key: str,  
    max_requests: int,  
    window_seconds: int  
) -> int:
```

```
    now = time.time()  
    window_start = now - window_seconds
```

```
    recent = [t for t in self.requests[key] if t > window_start]  
    return max(0, max_requests - len(recent))
```

```
class TieredRateLimiter:
```

```
    def __init__(self):  
        self.limiters = {  
            "global": RateLimiter(),  
            "user": RateLimiter(),  
            "ip": RateLimiter()  
        }
```

```
        self.limits = {  
            "global": (10000, 60),  
            "user": (60, 60),  
            "ip": (100, 60)  
        }
```

```
    def check_all_limits(  
        self,  
        user_id: str,  
        ip_address: str  
    ) -> tuple:
```

```
        global_ok = self.limiters["global"].is_allowed(  
            "global",  
            *self.limits["global"]  
        )
```

```

user_ok = self.limiters["user"].is_allowed(
    f"user:{user_id}",
    *self.limits["user"]
)

ip_ok = self.limiters["ip"].is_allowed(
    f"ip:{ip_address}",
    *self.limits["ip"]
)

return global_ok and user_ok and ip_ok, {
    "global": global_ok,
    "user": user_ok,
    "ip": ip_ok
}

```

SECTION 3: SECRETS PROTECTION

Environment Variable Security

```

import os
from typing import Optional

class SecureConfig:

    def __init__(self):
        self._secrets: Dict[str, str] = {}
        self._load_secrets()

    def _load_secrets(self):
        secret_vars = [
            "DATABASE_URL",
            "REDIS_URL",
            "TELEGRAM_BOT_TOKEN",
            "ETH_PRIVATE_KEY",
            "JWT_SECRET",
            "ENCRYPTION_KEY"
        ]

```



```
for var in secret_vars:
    value = os.environ.get(var)
    if value:
        self._secrets[var] = value
    os.environ[var] = "***REDACTED***"
```

```
def get(self, key: str) -> Optional[str]:
    return self._secrets.get(key)
```

```
def get_required(self, key: str) -> str:
    value = self._secrets.get(key)
    if not value:
        raise ValueError(f'Required secret {key} not found')
    return value
```

Private Key Protection

```
from cryptography.fernet import Fernet
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.kdf.pbkdf2 import
PBKDF2HMAC
import base64
```

```
class KeyEncryption:
```

```
    def __init__(self, master_password: str, salt: bytes):
        kdf = PBKDF2HMAC(
            algorithm=hashes.SHA256(),
            length=32,
            salt=salt,
            iterations=480000
        )
        key =
        base64.urlsafe_b64encode(kdf.derive(master_password.encode()))
        self.fernet = Fernet(key)
```

```
    def encrypt_key(self, private_key: str) -> bytes:
```

```
return self.fernet.encrypt(private_key.encode())
```

```
def decrypt_key(self, encrypted_key: bytes) -> str:  
return self.fernet.decrypt(encrypted_key).decode()
```

```
class SecureKeyStore:
```

```
def __init__(self, encryption: KeyEncryption, storage_path: str):  
self.encryption = encryption  
self.storage_path = storage_path
```

```
def store_key(self, key_id: str, private_key: str):  
encrypted = self.encryption.encrypt_key(private_key)
```

```
file_path = os.path.join(self.storage_path, f'{key_id}.key')
```

```
with open(file_path, "wb") as f:  
f.write(encrypted)
```

```
os.chmod(file_path, 0o600)
```

```
def load_key(self, key_id: str) -> str:  
file_path = os.path.join(self.storage_path, f'{key_id}.key')
```

```
with open(file_path, "rb") as f:  
encrypted = f.read()
```

```
return self.encryption.decrypt_key(encrypted)
```

SECTION 4: TLS CONFIGURATION

HTTPS Hardening

```
class TLSConfiguration:
```

```
@staticmethod  
def nginx_ssl_config() -> str:  
return """
```

```

ssl_protocols TLSv1.2 TLSv1.3;
ssl_ciphers ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-
AES128-GCM-SHA256:ECDHE-ECDSA-AES256-GCM-
SHA384:ECDHE-RSA-AES256-GCM-SHA384;
ssl_prefer_server_ciphers off;

ssl_session_timeout 1d;
ssl_session_cache shared:SSL:50m;
ssl_session_tickets off;

ssl_stapling on;
ssl_stapling_verify on;
resolver 8.8.8.8 8.8.4.4 valid = 300s;
resolver_timeout 5s;

add_header Strict-Transport-Security "max-age = 63072000" always;
""

@staticmethod
def security_headers() -> Dict[str, str]:
    return {
        "Strict-Transport-Security": "max-age = 63072000;
includeSubDomains",
        "X-Content-Type-Options": "nosniff",
        "X-Frame-Options": "DENY",
        "X-XSS-Protection": "1; mode = block",
        "Content-Security-Policy": "default-src 'self'",
        "Referrer-Policy": "strict-origin-when-cross-origin"
    }

```

SECTION 5: DATABASE SECURITY

Database Hardening

```

class DatabaseSecurityConfig:

    @staticmethod
    def generate_postgres_config() -> str:
        return ""

```

```

ssl = on
ssl_cert_file = '/etc/ssl/certs/server.crt'
ssl_key_file = '/etc/ssl/private/server.key'
ssl_ca_file = '/etc/ssl/certs/ca.crt'

password_encryption = scram-sha-256

log_connections = on
log_disconnections = on
log_statement = 'ddl'

shared_preload_libraries = 'pg_stat_statements'
"""

@staticmethod
def generate_pg_hba_rules() -> str:
    return """
local all postgres peer
local all all peer
hostssl all all 10.0.0.0/8 scram-sha-256
hostssl replication replicator 10.0.0.0/8 scram-sha-256
"""

class QuerySanitizer:

    @staticmethod
    def parameterized_query(query: str, params: Dict) -> tuple:
        return query, params

    @staticmethod
    def validate_table_name(table: str) -> bool:
        return bool(re.match(r'^[a-z_][a-z0-9_]*$', table))

```

SECTION 6: CONTAINER SECURITY

Docker Security

```

class DockerSecurityConfig:

```

```
@staticmethod
def secure_dockerfile() -> str:
    return """
FROM python:3.11-slim AS base
```

```
RUN apt-get update && apt-get upgrade -y && rm -rf /var/lib/apt/
lists/*
```

```
RUN groupadd -r appgroup && useradd -r -g appgroup appuser
```

```
WORKDIR /app
```

```
COPY --chown=appuser:appgroup requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

```
COPY --chown=appuser:appgroup . .
```

```
USER appuser
```

```
HEALTHCHECK --interval=30s --timeout=10s --retries=3 CMD
curl -f http://localhost:8000/health || exit 1
```

```
EXPOSE 8000
```

```
CMD ["gunicorn", "app:app", "-b", "0.0.0.0:8000"]
"""
```

```
@staticmethod
def docker_compose_security() -> str:
    return """
version: '3.8'
```

```
services:
  app:
    build: .
    read_only: true
    tmpfs:
      - /tmp
    security_opt:
```

```
- no-new-privileges:true
cap_drop:
- ALL
cap_add:
- NET_BIND_SERVICE
user: "1000:1000"
deploy:
resources:
limits:
cpus: '2'
memory: 1G
""""
```

Kubernetes Security

```
apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    runAsGroup: 1000
    fsGroup: 1000
  containers:
  - name: app
    image: app:latest
    securityContext:
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
    capabilities:
      drop:
      - ALL
    resources:
      limits:
        cpu: "1"
        memory: "512Mi"
      requests:
```

```
cpu: "100m"  
memory: "128Mi"  
volumeMounts:  
- name: tmp  
mountPath: /tmp  
volumes:  
- name: tmp  
emptyDir: {}
```

SECTION 7: AUDIT LOGGING

Security Audit Logger

```
import json  
from datetime import datetime  
from typing import Optional
```

```
class SecurityAuditLogger:
```

```
    def __init__(self, log_path: str):  
        self.log_path = log_path
```

```
    def log_event(  
        self,  
        event_type: str,  
        user_id: Optional[str],  
        ip_address: str,  
        action: str,  
        resource: str,  
        status: str,  
        details: Dict = None  
    ):  
        event = {  
            "timestamp": datetime.utcnow().isoformat() + "Z",  
            "event_type": event_type,  
            "user_id": user_id,  
            "ip_address": ip_address,  
            "action": action,
```

```
"resource": resource,  
"status": status,  
"details": details or {}  
}
```

```
with open(self.log_path, "a") as f:  
f.write(json.dumps(event) + "\n")
```

```
def log_authentication(  
self,  
user_id: str,  
ip_address: str,  
success: bool,  
method: str  
):  
self.log_event(  
event_type="authentication",  
user_id=user_id,  
ip_address=ip_address,  
action="login",  
resource="auth",  
status="success" if success else "failure",  
details={"method": method}  
)
```

```
def log_transaction(  
self,  
user_id: str,  
ip_address: str,  
tx_type: str,  
amount: int,  
address: str  
):  
self.log_event(  
event_type="transaction",  
user_id=user_id,  
ip_address=ip_address,  
action=tx_type,  
resource="blockchain",  
status="initiated",
```



```

details = {"amount": amount, "address": address}
)

def log_admin_action(
    self,
    admin_id: str,
    ip_address: str,
    action: str,
    target: str
):
    self.log_event(
        event_type="admin",
        user_id=admin_id,
        ip_address=ip_address,
        action=action,
        resource=target,
        status="executed",
        details={}
    )

```

This chapter covered security hardening for production systems. The next chapter covers maintenance and operations procedures.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 6: Maintenance and Operations

Production systems need constant care. Updates. Backups. Performance tuning. This chapter teaches you to maintain crypto and Telegram applications reliably.

SECTION 1: BACKUP STRATEGIES

Database Backups

```

import subprocess
from datetime import datetime
from typing import Optional
import os

```

```
class DatabaseBackupManager:
```

```
    def __init__(
        self,
        db_host: str,
        db_name: str,
        db_user: str,
        backup_path: str
    ):
        self.db_host = db_host
        self.db_name = db_name
        self.db_user = db_user
        self.backup_path = backup_path
```

```
    def create_backup(self) -> str:
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f'{self.db_name}_{timestamp}.sql.gz'
        filepath = os.path.join(self.backup_path, filename)
```

```
        cmd = f'pg_dump -h {self.db_host} -U {self.db_user} -d
{self.db_name} | gzip > {filepath}'
```

```
        result = subprocess.run(cmd, shell=True, capture_output=True)
```

```
        if result.returncode != 0:
            raise Exception(f'Backup failed: {result.stderr.decode()}')
```

```
        return filepath
```

```
    def restore_backup(self, backup_file: str):
        cmd = f'gunzip -c {backup_file} | psql -h {self.db_host} -U
{self.db_user} -d {self.db_name}'
```

```
        result = subprocess.run(cmd, shell=True, capture_output=True)
```

```
        if result.returncode != 0:
            raise Exception(f'Restore failed: {result.stderr.decode()}')
```

```
    def cleanup_old_backups(self, keep_days: int = 30):
```

```

import glob
import time

cutoff = time.time() - (keep_days * 86400)
pattern = os.path.join(self.backup_path, "*.sql.gz")

for filepath in glob.glob(pattern):
    if os.path.getmtime(filepath) < cutoff:
        os.remove(filepath)

```

Wallet Backup

```

class WalletBackupManager:

    def __init__(self, encryption_key: str, backup_path: str):
        self.encryption_key = encryption_key
        self.backup_path = backup_path

    def backup_wallet_data(self, wallet_data: Dict) -> str:
        from cryptography.fernet import Fernet
        import json

        fernet = Fernet(self.encryption_key.encode())

        data_json = json.dumps(wallet_data)
        encrypted = fernet.encrypt(data_json.encode())

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        filename = f"wallet_backup_{timestamp}.enc"
        filepath = os.path.join(self.backup_path, filename)

        with open(filepath, "wb") as f:
            f.write(encrypted)

        os.chmod(filepath, 0o600)

        return filepath

    def restore_wallet_data(self, backup_file: str) -> Dict:

```

```
from cryptography.fernet import Fernet
import json

fernet = Fernet(self.encrypted_key.encode())

with open(backup_file, "rb") as f:
    encrypted = f.read()

decrypted = fernet.decrypt(encrypted)
return json.loads(decrypted.decode())
```

SECTION 2: UPDATE PROCEDURES

Application Updates

```
class ApplicationUpdateManager:

    def __init__(self, app_path: str, backup_path: str):
        self.app_path = app_path
        self.backup_path = backup_path

    def pre_update_checks(self) -> Dict[str, bool]:
        checks = {}

        checks["backup_recent"] = self._check_recent_backup()
        checks["disk_space"] = self._check_disk_space()
        checks["no_active_transactions"] =
self._check_no_active_transactions()
        checks["health_check_passing"] = self._check_health()

    return checks

    def _check_recent_backup(self) -> bool:
        import glob
        import time

        pattern = os.path.join(self.backup_path, "*.sql.gz")
        backups = glob.glob(pattern)
```

```
if not backups:  
    return False
```

```
latest = max(backups, key=os.path.getmtime)  
age = time.time() - os.path.getmtime(latest)
```

```
return age < 3600
```

```
def _check_disk_space(self) -> bool:  
    import shutil  
    total, used, free = shutil.disk_usage(self.app_path)  
    return free > 1e9
```

```
def _check_no_active_transactions(self) -> bool:  
    return True
```

```
def _check_health(self) -> bool:  
    return True
```

```
def perform_update(  
    self,  
    new_version: str,  
    migration_func: Optional[callable] = None  
) -> Dict:
```

```
    result = {  
        "success": False,  
        "old_version": self._get_current_version(),  
        "new_version": new_version,  
        "steps": []  
    }
```

```
    try:  
        result["steps"].append(("backup", self._create_backup()))  
        result["steps"].append(("stop_services", self._stop_services()))  
        result["steps"].append(("update_code",  
            self._update_code(new_version)))
```

```
    if migration_func:  
        result["steps"].append(("migration", migration_func()))
```

```
result["steps"].append(("start_services", self._start_services()))
result["steps"].append(("health_check", self._verify_health()))
```

```
result["success"] = True
```

```
except Exception as e:
    result["error"] = str(e)
    self._rollback()
```

```
return result
```

```
def _get_current_version(self) -> str:
    return "1.0.0"
```

```
def _create_backup(self) -> bool:
    return True
```

```
def _stop_services(self) -> bool:
    return True
```

```
def _update_code(self, version: str) -> bool:
    return True
```

```
def _start_services(self) -> bool:
    return True
```

```
def _verify_health(self) -> bool:
    return True
```

```
def _rollback(self):
    pass
```

Database Migrations

```
class MigrationManager:
```

```
    def __init__(self, db_url: str, migrations_path: str):
        self.db_url = db_url
```

```
self.migrations_path = migrations_path
```

```
def get_pending_migrations(self) -> List[str]:  
    return []
```

```
def run_migrations(self) -> Dict:  
    result = subprocess.run(  
        ["alembic", "upgrade", "head"],  
        capture_output=True,  
        text=True,  
        env={**os.environ, "DATABASE_URL": self.db_url}  
    )
```

```
    return {  
        "success": result.returncode == 0,  
        "output": result.stdout,  
        "error": result.stderr  
    }
```

```
def rollback_migration(self, steps: int = 1) -> Dict:  
    result = subprocess.run(  
        ["alembic", "downgrade", f"-{steps}"],  
        capture_output=True,  
        text=True,  
        env={**os.environ, "DATABASE_URL": self.db_url}  
    )
```

```
    return {  
        "success": result.returncode == 0,  
        "output": result.stdout,  
        "error": result.stderr  
    }
```

SECTION 3: PERFORMANCE TUNING

Database Optimization

```
class DatabaseOptimizer:
```

```
def __init__(self, db_connection):
    self.db = db_connection
```

```
async def analyze_slow_queries(self) -> List[Dict]:
    query = """
    SELECT
    query,
    calls,
    total_time / 1000 as total_seconds,
    mean_time / 1000 as mean_seconds,
    rows
    FROM pg_stat_statements
    WHERE total_time > 1000
    ORDER BY total_time DESC
    LIMIT 20
    """
```

```
return await self.db.fetch_all(query)
```

```
async def get_index_recommendations(self) -> List[Dict]:
    query = """
    SELECT
    schemaname,
    relname,
    seq_scan,
    seq_tup_read,
    idx_scan,
    idx_tup_fetch
    FROM pg_stat_user_tables
    WHERE seq_scan > 100 AND seq_tup_read > 10000
    ORDER BY seq_tup_read DESC
    """
```

```
return await self.db.fetch_all(query)
```

```
async def vacuum_analyze(self, table: str = None):
    if table:
        await self.db.execute(f"VACUUM ANALYZE {table}")
    else:
        await self.db.execute("VACUUM ANALYZE")
```


Application Profiling

```
import cProfile
import pstats
import io
```

```
class ApplicationProfiler:
```

```
    def __init__(self):
        self.profiler = None
```

```
    def start_profiling(self):
        self.profiler = cProfile.Profile()
        self.profiler.enable()
```

```
    def stop_profiling(self) -> str:
        if not self.profiler:
            return ""
```

```
        self.profiler.disable()
```

```
        stream = io.StringIO()
        stats = pstats.Stats(self.profiler, stream = stream)
        stats.sort_stats("cumulative")
        stats.print_stats(50)
```

```
        return stream.getvalue()
```

SECTION 4: LOG MANAGEMENT

Log Rotation

```
class LogManager:
```

```
    def __init__(self, log_dir: str):
        self.log_dir = log_dir
```

```

def configure_rotation(self) -> str:
    return f"""
{self.log_dir}/*.log {{
    daily
    rotate 30
    compress
    delaycompress
    missingok
    notifempty
    create 0640 app app
    sharedscripts
    postrotate
    systemctl reload app 2>/dev/null || true
    endsript
}}
"""

```

```

def cleanup_old_logs(self, max_age_days: int = 30):
    import glob
    import time

```

```

    cutoff = time.time() - (max_age_days * 86400)

```

```

    for pattern in ["*.log", "*.log.gz", "*.log.*.gz"]:
        for filepath in glob.glob(os.path.join(self.log_dir, pattern)):
            if os.path.getmtime(filepath) < cutoff:
                os.remove(filepath)

```

Log Aggregation

```

class LogAggregator:

```

```

    def __init__(self, elasticsearch_url: str):
        self.es_url = elasticsearch_url

```

```

    def generate_filebeat_config(self, log_paths: List[str]) -> str:
        inputs = []
        for path in log_paths:

```

```

inputs.append(f"""
- type: log
enabled: true
paths:
- {path}
json.keys_under_root: true
json.add_error_key: true
""")

return f"""
filebeat.inputs:
{".join(inputs)}

output.elasticsearch:
hosts: ["{self.es_url}"]
index: "app-logs-%{{ + yyyy.MM.dd }}"

setup.ilm.enabled: true
setup.ilm.rollover_alias: "app-logs"
setup.ilm.pattern: "{now/d}}-000001"
"""

```

SECTION 5: SCHEDULED TASKS

Task Scheduler

```

import asyncio
from dataclasses import dataclass
from typing import Callable, Awaitable

```

```

@dataclass
class ScheduledTask:
    name: str
    func: Callable[[], Awaitable[None]]
    interval_seconds: int
    last_run: float = 0
    enabled: bool = True

```

```
class TaskScheduler:

    def __init__(self):
        self.tasks: Dict[str, ScheduledTask] = {}
        self.running = False

    def add_task(
        self,
        name: str,
        func: Callable,
        interval_seconds: int
    ):
        self.tasks[name] = ScheduledTask(
            name=name,
            func=func,
            interval_seconds=interval_seconds
        )

    async def run(self):
        self.running = True

        while self.running:
            now = time.time()

            for task in self.tasks.values():
                if not task.enabled:
                    continue

            if now - task.last_run >= task.interval_seconds:
                try:
                    await task.func()
                    task.last_run = now
                except Exception as e:
                    print(f"Task {task.name} failed: {e}")

            await asyncio.sleep(1)

    def stop(self):
        self.running = False
```

Maintenance Tasks

```
class MaintenanceTasks:
```

```
    def __init__(
        self,
        db_backup: DatabaseBackupManager,
        db_optimizer: DatabaseOptimizer,
        log_manager: LogManager
    ):
        self.db_backup = db_backup
        self.db_optimizer = db_optimizer
        self.log_manager = log_manager
```

```
    async def daily_backup(self):
        self.db_backup.create_backup()
```

```
    async def hourly_vacuum(self):
        await self.db_optimizer.vacuum_analyze()
```

```
    async def weekly_cleanup(self):
        self.db_backup.cleanup_old_backups(keep_days=30)
        self.log_manager.cleanup_old_logs(max_age_days=30)
```

```
    def register_all(self, scheduler: TaskScheduler):
        scheduler.add_task("daily_backup", self.daily_backup, 86400)
        scheduler.add_task("hourly_vacuum", self.hourly_vacuum, 3600)
        scheduler.add_task("weekly_cleanup", self.weekly_cleanup, 604800)
```

SECTION 6: RUNBOOKS

Operational Runbooks

```
class OperationalRunbooks:
```

```
    @staticmethod
    def restart_application() -> str:
```

```
return ""
```

APPLICATION RESTART PROCEDURE

1. Check current status
systemctl status app
2. Graceful shutdown
systemctl stop app
3. Wait for connections to drain
sleep 10
4. Start application
systemctl start app
5. Verify health
curl http://localhost:8000/health
6. Monitor logs
journalctl -u app -f

```
@staticmethod
```

```
def database_failover() -> str:
```

```
return ""
```

DATABASE FAILOVER PROCEDURE

1. Verify primary is down
pg_isready -h primary.db -p 5432
2. Promote replica
sudo -u postgres pg_ctl promote -D /var/lib/postgresql/data
3. Update application config
Update DATABASE_URL to point to new primary
4. Restart application
systemctl restart app
5. Verify connectivity

```
psql -h new-primary.db -U app -d appdb -c "SELECT 1"
```

6. Set up new replica from promoted primary

```
"""
```

```
@staticmethod
```

```
def clear_cache() -> str:
```

```
    return """
```

```
CACHE CLEAR PROCEDURE
```

1. Check cache usage

```
redis-cli INFO memory
```

2. Clear specific keys

```
redis-cli KEYS "cache:*" | xargs redis-cli DEL
```

3. Or flush entire database

```
redis-cli FLUSHDB
```

4. Verify cache is cleared

```
redis-cli DBSIZE
```

5. Monitor application for cache misses

```
    Watch metrics for increased database load
```

```
"""
```

This chapter covered maintenance and operations procedures. The next chapter covers disaster recovery planning.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 7: Disaster Recovery

Systems fail. Hardware dies. Data centers flood. Regions go dark. This chapter teaches you to plan for the worst and recover from disasters affecting your crypto and Telegram infrastructure.

SECTION 1: DISASTER RECOVERY PLANNING

Recovery Objectives

class RecoveryObjectives:

```
def __init__(self):
    self.rto_targets = {
        "critical": 15,
        "high": 60,
        "medium": 240,
        "low": 1440
    }
```

```
    self.rpo_targets = {
        "critical": 0,
        "high": 5,
        "medium": 60,
        "low": 1440
    }
```

```
def classify_service(self, service_name: str) -> Dict:
    classifications = {
        "payment_processing": {
            "tier": "critical",
            "rto_minutes": 15,
            "rpo_minutes": 0,
            "justification": "Financial transactions cannot be lost"
        },
        "wallet_service": {
            "tier": "critical",
            "rto_minutes": 15,
            "rpo_minutes": 0,
            "justification": "User funds must be protected"
        },
        "telegram_bot": {
            "tier": "high",
            "rto_minutes": 60,
            "rpo_minutes": 5,
            "justification": "User communication channel"
        },
        "price_feeds": {
            "tier": "high",
```



```

"rto_minutes": 60,
"rpo_minutes": 5,
"justification": "Trading decisions depend on prices"
},
"analytics": {
"tier": "medium",
"rto_minutes": 240,
"rpo_minutes": 60,
"justification": "Non-critical reporting"
},
"notifications": {
"tier": "low",
"rto_minutes": 1440,
"rpo_minutes": 1440,
"justification": "Can be delayed without impact"
}
}

return classifications.get(service_name, {
"tier": "medium",
"rto_minutes": 240,
"rpo_minutes": 60,
"justification": "Default classification"
})

```

Disaster Scenarios

```
class DisasterScenario:
```

```

    def __init__(self, name: str, probability: str, impact: str):
        self.name = name
        self.probability = probability
        self.impact = impact
        self.mitigation_steps = []
        self.recovery_steps = []

    def add_mitigation(self, step: str):
        self.mitigation_steps.append(step)

```

```
def add_recovery(self, step: str):  
    self.recovery_steps.append(step)
```

```
class DisasterCatalog:
```

```
    @staticmethod  
    def get_scenarios() -> List[DisasterScenario]:  
        scenarios = []
```

```
        db_failure = DisasterScenario(  
            name="Primary Database Failure",  
            probability="medium",  
            impact="critical"  
        )  
        db_failure.add_mitigation("Synchronous replication to standby")  
        db_failure.add_mitigation("Automated failover with health checks")  
        db_failure.add_mitigation("Regular backup verification")  
        db_failure.add_recovery("Promote standby to primary")  
        db_failure.add_recovery("Update connection strings")  
        db_failure.add_recovery("Verify data consistency")  
        db_failure.add_recovery("Provision new standby")  
        scenarios.append(db_failure)
```

```
        region_outage = DisasterScenario(  
            name="Cloud Region Outage",  
            probability="low",  
            impact="critical"  
        )  
        region_outage.add_mitigation("Multi-region deployment")  
        region_outage.add_mitigation("Cross-region data replication")  
        region_outage.add_mitigation("DNS-based failover")  
        region_outage.add_recovery("Activate secondary region")  
        region_outage.add_recovery("Update DNS records")  
        region_outage.add_recovery("Scale secondary region resources")  
        region_outage.add_recovery("Notify users of degraded service")  
        scenarios.append(region_outage)
```

```
        key_compromise = DisasterScenario(  
            name="Private Key Compromise",
```

```

probability = "low",
impact = "critical"
)
key_compromise.add_mitigation("Multi-signature wallets")
key_compromise.add_mitigation("Hardware security modules")
key_compromise.add_mitigation("Key rotation procedures")
key_compromise.add_recovery("Freeze affected wallets")
key_compromise.add_recovery("Transfer funds to new wallets")
key_compromise.add_recovery("Rotate all related secrets")
key_compromise.add_recovery("Conduct security audit")
scenarios.append(key_compromise)

```

```

ransomware = DisasterScenario(
name = "Ransomware Attack",
probability = "medium",
impact = "critical"
)
ransomware.add_mitigation("Immutable backups")
ransomware.add_mitigation("Network segmentation")
ransomware.add_mitigation("Endpoint detection")
ransomware.add_recovery("Isolate infected systems")
ransomware.add_recovery("Restore from clean backups")
ransomware.add_recovery("Rebuild affected infrastructure")
ransomware.add_recovery("Forensic investigation")
scenarios.append(ransomware)

```

```

return scenarios

```

SECTION 2: BACKUP ARCHITECTURE

Multi-Tier Backup Strategy

```

import boto3
from datetime import datetime, timedelta
import hashlib

```

```

class BackupTier:

```

```
def __init__(self, name: str, retention_days: int, storage_class: str):
    self.name = name
    self.retention_days = retention_days
    self.storage_class = storage_class
```

```
class MultiTierBackupManager:
```

```
    def __init__(self, s3_client, bucket_name: str):
        self.s3 = s3_client
        self.bucket = bucket_name
```

```
        self.tiers = {
            "hot": BackupTier("hot", 7, "STANDARD"),
            "warm": BackupTier("warm", 30, "STANDARD_IA"),
            "cold": BackupTier("cold", 365, "GLACIER"),
            "archive": BackupTier("archive", 2555, "DEEP_ARCHIVE")
        }
```

```
    def upload_backup(
        self,
        backup_data: bytes,
        backup_name: str,
        tier: str = "hot"
    ) -> Dict:
```

```
        tier_config = self.tiers[tier]
        timestamp = datetime.utcnow().strftime("%Y%m%d_%H%M%S")
        key = f"backups/{tier}/{backup_name}_{timestamp}"
```

```
        checksum = hashlib.sha256(backup_data).hexdigest()
```

```
        self.s3.put_object(
            Bucket=self.bucket,
            Key=key,
            Body=backup_data,
            StorageClass=tier_config.storage_class,
            Metadata={
                "checksum": checksum,
                "backup_name": backup_name,
```

```
"tier": tier,  
"created_at": timestamp  
}  
)
```

```
return {  
"key": key,  
"checksum": checksum,  
"tier": tier,  
"storage_class": tier_config.storage_class  
}
```

```
def promote_backup(self, source_key: str, target_tier: str) -> str:  
    tier_config = self.tiers[target_tier]
```

```
    copy_source = {"Bucket": self.bucket, "Key": source_key}  
    target_key = source_key.replace(  
        f'backups/{self._get_current_tier(source_key)}/',  
        f'backups/{target_tier}/'  
    )
```

```
    self.s3.copy_object(  
        Bucket=self.bucket,  
        CopySource=copy_source,  
        Key=target_key,  
        StorageClass=tier_config.storage_class  
    )
```

```
    return target_key
```

```
def _get_current_tier(self, key: str) -> str:  
    parts = key.split("/")  
    if len(parts) >= 2:  
        return parts[1]  
    return "hot"
```

```
def apply_lifecycle_rules(self):  
    lifecycle_config = {  
        "Rules": [  
            {
```

```

"ID": "hot-to-warm",
"Status": "Enabled",
"Filter": {"Prefix": "backups/hot/"},
"Transitions": [
{"Days": 7, "StorageClass": "STANDARD_IA"}
],
{
"ID": "warm-to-cold",
"Status": "Enabled",
"Filter": {"Prefix": "backups/warm/"},
"Transitions": [
{"Days": 30, "StorageClass": "GLACIER"}
],
{
"ID": "cold-to-archive",
"Status": "Enabled",
"Filter": {"Prefix": "backups/cold/"},
"Transitions": [
{"Days": 365, "StorageClass": "DEEP_ARCHIVE"}
],
{
"ID": "archive-expiration",
"Status": "Enabled",
"Filter": {"Prefix": "backups/archive/"},
"Expiration": {"Days": 2555}
}
]
}

```

```

self.s3.put_bucket_lifecycle_configuration(
    Bucket=self.bucket,
    LifecycleConfiguration=lifecycle_config
)

```

Cross-Region Replication

```
class CrossRegionBackup:
```

```
    def __init__(
        self,
        primary_region: str,
        secondary_region: str,
        bucket_name: str
    ):
        self.primary = boto3.client("s3", region_name=primary_region)
        self.secondary = boto3.client("s3",
            region_name=secondary_region)
        self.primary_bucket = f'{bucket_name}-{primary_region}'
        self.secondary_bucket = f'{bucket_name}-{secondary_region}'
```

```
    def setup_replication(self):
        replication_config = {
            "Role": "arn:aws:iam::ACCOUNT:role/s3-replication-role",
            "Rules": [
                {
                    "ID": "backup-replication",
                    "Status": "Enabled",
                    "Priority": 1,
                    "Filter": {"Prefix": "backups/"},
                    "Destination": {
                        "Bucket": f'arn:aws:s3:::{self.secondary_bucket}',
                        "StorageClass": "STANDARD_IA",
                        "ReplicationTime": {
                            "Status": "Enabled",
                            "Time": {"Minutes": 15}
                        },
                    },
                    "Metrics": {
                        "Status": "Enabled",
                        "EventThreshold": {"Minutes": 15}
                    },
                },
                {
                    "DeleteMarkerReplication": {"Status": "Disabled"}
                }
            ]
        }
```

```

self.primary.put_bucket_replication(
    Bucket=self.primary_bucket,
    ReplicationConfiguration=replication_config
)

def verify_replication(self, backup_key: str) -> bool:
    try:
        primary_obj = self.primary.head_object(
            Bucket=self.primary_bucket,
            Key=backup_key
        )

        secondary_obj = self.secondary.head_object(
            Bucket=self.secondary_bucket,
            Key=backup_key
        )

        return (
            primary_obj["ContentLength"] == secondary_obj["ContentLength"]
            and primary_obj["ETag"] == secondary_obj["ETag"]
        )

    except Exception:
        return False

```

SECTION 3: FAILOVER MECHANISMS

Automated Failover Controller

```

import asyncio
from enum import Enum
from dataclasses import dataclass

class FailoverState(Enum):
    NORMAL = "normal"
    DEGRADED = "degraded"
    FAILOVER_INITIATED = "failover_initiated"
    FAILOVER_COMPLETE = "failover_complete"

```



```
RECOVERY = "recovery"
```

```
@dataclass
class ServiceEndpoint:
    name: str
    primary_url: str
    secondary_url: str
    health_check_path: str
    is_primary_active: bool = True
```

```
class FailoverController:
```

```
    def __init__(self):
        self.endpoints: Dict[str, ServiceEndpoint] = {}
        self.state = FailoverState.NORMAL
        self.health_history: Dict[str, List[bool]] = {}
        self.failure_threshold = 3
        self.recovery_threshold = 5
```

```
    def register_endpoint(self, endpoint: ServiceEndpoint):
        self.endpoints[endpoint.name] = endpoint
        self.health_history[endpoint.name] = []
```

```
    async def check_health(self, endpoint: ServiceEndpoint) -> bool:
        import aiohttp
```

```
        url = (
            endpoint.primary_url if endpoint.is_primary_active
            else endpoint.secondary_url
        )
        health_url = f"{url}{endpoint.health_check_path}"
```

```
        try:
            async with aiohttp.ClientSession() as session:
                async with session.get(health_url, timeout=5) as response:
                    return response.status == 200
        except Exception:
            return False
```

```

async def monitor_loop(self):
    while True:
        for name, endpoint in self.endpoints.items():
            healthy = await self.check_health(endpoint)
            self.health_history[name].append(healthy)
            self.health_history[name] = self.health_history[name][-10:]

            recent_failures = self.health_history[name][-self.failure_threshold:]
            if all(not h for h in recent_failures) and endpoint.is_primary_active:
                await self.initiate_failover(name)

            recent_successes = self.health_history[name][-
self.recovery_threshold:]
            if (all(recent_successes) and
                not endpoint.is_primary_active and
                self.state == FailoverState.FAILOVER_COMPLETE):
                await self.initiate_recovery(name)

        await asyncio.sleep(10)

    async def initiate_failover(self, service_name: str):
        self.state = FailoverState.FAILOVER_INITIATED

        endpoint = self.endpoints[service_name]
        endpoint.is_primary_active = False

        await self._update_load_balancer(endpoint)
        await self._send_alert(f'Failover initiated for {service_name}')

        self.state = FailoverState.FAILOVER_COMPLETE

    async def initiate_recovery(self, service_name: str):
        self.state = FailoverState.RECOVERY

        endpoint = self.endpoints[service_name]

        if await self._verify_primary_health(endpoint):
            endpoint.is_primary_active = True
            await self._update_load_balancer(endpoint)

```

```
await self._send_alert(f'Recovery complete for {service_name}')
self.state = FailoverState.NORMAL
```

```
async def _update_load_balancer(self, endpoint: ServiceEndpoint):
    pass
```

```
async def _verify_primary_health(self, endpoint: ServiceEndpoint) -
> bool:
    return True
```

```
async def _send_alert(self, message: str):
    pass
```

Database Failover

```
class DatabaseFailoverManager:
```

```
    def __init__(
        self,
        primary_host: str,
        replica_hosts: List[str],
        db_name: str,
        db_user: str
    ):
        self.primary_host = primary_host
        self.replica_hosts = replica_hosts
        self.db_name = db_name
        self.db_user = db_user
        self.current_primary = primary_host
```

```
    async def check_primary_health(self) -> bool:
        import asyncpg
```

```
        try:
            conn = await asyncpg.connect(
                host=self.current_primary,
                database=self.db_name,
                user=self.db_user,
                timeout=5
```

```

)
await conn.fetchval("SELECT 1")
await conn.close()
return True
except Exception:
return False

```

```

async def get_replica_lag(self, replica_host: str) -> int:
import asyncpg

```

```

try:
conn = await asyncpg.connect(
host=replica_host,
database=self.db_name,
user=self.db_user,
timeout=5
)

```

```

result = await conn.fetchrow("""
SELECT
EXTRACT(EPOCH FROM (now() -
pg_last_xact_replay_timestamp()))::int
AS lag_seconds
""")

```

```

await conn.close()
return result["lag_seconds"] if result["lag_seconds"] else 0

```

```

except Exception:
return -1

```

```

async def select_best_replica(self) -> str:
best_replica = None
lowest_lag = float("inf")

```

```

for replica in self.replica_hosts:
lag = await self.get_replica_lag(replica)
if 0 <= lag < lowest_lag:
lowest_lag = lag
best_replica = replica

```

```
return best_replica
```

```
async def promote_replica(self, replica_host: str) -> bool:  
import asyncpg
```

```
try:  
    conn = await asyncpg.connect(  
        host=replica_host,  
        database=self.db_name,  
        user=self.db_user  
    )
```

```
    await conn.execute("SELECT pg_promote()")  
    await conn.close()
```

```
    self.current_primary = replica_host  
    self.replica_hosts.remove(replica_host)
```

```
    return True
```

```
except Exception:  
    return False
```

```
async def automated_failover(self) -> Dict:  
    result = {  
        "success": False,  
        "old_primary": self.current_primary,  
        "new_primary": None,  
        "steps": []  
    }
```

```
    if await self.check_primary_health():  
        result["steps"].append("Primary is healthy, no failover needed")  
    return result
```

```
    result["steps"].append("Primary unhealthy, selecting best replica")
```

```
    best_replica = await self.select_best_replica()  
    if not best_replica:
```

```

result["steps"].append("No healthy replica available")
return result

result["steps"].append(f"Selected replica: {best_replica}")

if await self.promote_replica(best_replica):
    result["success"] = True
    result["new_primary"] = best_replica
    result["steps"].append("Replica promoted successfully")
else:
    result["steps"].append("Replica promotion failed")

return result

```

SECTION 4: DATA RECOVERY

Point-in-Time Recovery

```

class PointInTimeRecovery:

    def __init__(
        self,
        backup_manager: MultiTierBackupManager,
        wal_archive_path: str
    ):
        self.backup_manager = backup_manager
        self.wal_archive_path = wal_archive_path

    def list_recovery_points(self, start_time: datetime, end_time:
datetime) -> List[Dict]:
        recovery_points = []

        response = self.backup_manager.s3.list_objects_v2(
            Bucket=self.backup_manager.bucket,
            Prefix="backups/"
        )

        for obj in response.get("Contents", []):
            metadata = self.backup_manager.s3.head_object(

```

```

Bucket = self.backup_manager.bucket,
Key = obj["Key"]
).get("Metadata", {})

created_at = metadata.get("created_at", "")
if created_at:
    backup_time = datetime.strptime(created_at, "%Y%m%d_%H%M%S")
    if start_time <= backup_time <= end_time:
        recovery_points.append({
            "key": obj["Key"],
            "timestamp": backup_time,
            "size": obj["Size"],
            "tier": metadata.get("tier", "unknown")
        })

return sorted(recovery_points, key=lambda x: x["timestamp"])

```

```

def generate_recovery_commands(
    self,
    backup_key: str,
    target_time: datetime,
    data_directory: str
) -> str:

```

```

    return f"""

```

POINT-IN-TIME RECOVERY PROCEDURE

1. Stop PostgreSQL

```
sudo systemctl stop postgresql
```

2. Backup current data directory

```
sudo mv {data_directory} {data_directory}.old
```

3. Download and extract base backup

```

aws s3 cp s3://{self.backup_manager.bucket}/{backup_key} /tmp/
base_backup.tar.gz
sudo mkdir -p {data_directory}
sudo tar -xzf /tmp/base_backup.tar.gz -C {data_directory}

```

4. Configure recovery

```
cat > {data_directory}/recovery.signal << EOF
EOF
```

```
cat >> {data_directory}/postgresql.conf << EOF
restore_command = 'aws s3 cp s3://
{self.backup_manager.bucket}/wal/%f %p'
recovery_target_time = '{target_time.isoformat()}'
recovery_target_action = 'promote'
EOF
```

5. Start PostgreSQL

```
sudo chown -R postgres:postgres {data_directory}
sudo systemctl start postgresql
```

6. Verify recovery

```
sudo -u postgres psql -c "SELECT pg_is_in_recovery();"
```

7. When recovery is complete

```
SELECT pg_wal_replay_resume();
""""
```

Wallet Recovery

```
class WalletRecoveryManager:
```

```
def __init__(self, backup_path: str, encryption_key: str):
    self.backup_path = backup_path
    self.encryption_key = encryption_key
```

```
def list_wallet_backups(self) -> List[Dict]:
    import glob
```

```
    backups = []
    pattern = os.path.join(self.backup_path, "wallet_backup_*.enc")
```

```
    for filepath in glob.glob(pattern):
        filename = os.path.basename(filepath)
        timestamp_str = filename.replace("wallet_backup_",
```



```
""").replace(".enc", "")
```

```
try:
    timestamp = datetime.strptime(timestamp_str, "%Y%m%d_%H%M%S")
    backups.append({
        "filepath": filepath,
        "timestamp": timestamp,
        "size": os.path.getsize(filepath)
    })
except ValueError:
    continue
```

```
return sorted(backups, key=lambda x: x["timestamp"],
reverse=True)
```

```
def verify_backup_integrity(self, backup_file: str) -> bool:
    from cryptography.fernet import Fernet, InvalidToken
```

```
try:
    fernet = Fernet(self.encrypted_key.encode())
```

```
with open(backup_file, "rb") as f:
    encrypted = f.read()
```

```
fernet.decrypt(encrypted)
return True
```

```
except (InvalidToken, Exception):
    return False
```

```
def recover_wallet(self, backup_file: str) -> Dict:
    from cryptography.fernet import Fernet
    import json
```

```
fernet = Fernet(self.encrypted_key.encode())
```

```
with open(backup_file, "rb") as f:
    encrypted = f.read()
```

```

decrypted = fernet.decrypt(encrypted)
wallet_data = json.loads(decrypted.decode())

return wallet_data

def generate_recovery_report(self, wallet_data: Dict) -> Dict:
    report = {
        "addresses_recovered": [],
        "total_addresses": 0,
        "has_private_keys": False,
        "has_mnemonics": False,
        "warnings": []
    }

    if "addresses" in wallet_data:
        report["addresses_recovered"] = wallet_data["addresses"]
        report["total_addresses"] = len(wallet_data["addresses"])

    if "private_keys" in wallet_data:
        report["has_private_keys"] = True

    if "mnemonic" in wallet_data:
        report["has_mnemonics"] = True

    if not report["has_private_keys"] and not report["has_mnemonics"]:
        report["warnings"].append("No recovery secrets found in backup")

    return report

```

SECTION 5: DISASTER RECOVERY TESTING

DR Test Framework

```

class DRTestScenario:

    def __init__(self, name: str, description: str):
        self.name = name
        self.description = description
        self.steps: List[Dict] = []

```

```
self.results: List[Dict] = []
```

```
def add_step(self, step_name: str, action: callable):  
    self.steps.append({  
        "name": step_name,  
        "action": action  
    })
```

```
class DRTestRunner:
```

```
    def __init__(self):  
        self.scenarios: List[DRTestScenario] = []  
        self.test_results: Dict[str, Dict] = {}
```

```
    def add_scenario(self, scenario: DRTestScenario):  
        self.scenarios.append(scenario)
```

```
    async def run_scenario(self, scenario: DRTestScenario) -> Dict:  
        result = {  
            "scenario": scenario.name,  
            "success": True,  
            "steps": [],  
            "start_time": datetime.utcnow(),  
            "end_time": None,  
            "duration_seconds": 0  
        }
```

```
        for step in scenario.steps:  
            step_result = {  
                "name": step["name"],  
                "success": False,  
                "error": None,  
                "duration_seconds": 0  
            }
```

```
            step_start = datetime.utcnow()
```

```
            try:  
                await step["action"]()
```

```
step_result["success"] = True
except Exception as e:
step_result["success"] = False
step_result["error"] = str(e)
result["success"] = False
```

```
step_result["duration_seconds"] = (
datetime.utcnow() - step_start
).total_seconds()
```

```
result["steps"].append(step_result)
```

```
if not step_result["success"]:
break
```

```
result["end_time"] = datetime.utcnow()
result["duration_seconds"] = (
result["end_time"] - result["start_time"]
).total_seconds()
```

```
self.test_results[scenario.name] = result
return result
```

```
async def run_all_scenarios(self) -> Dict:
all_results = {
"total_scenarios": len(self.scenarios),
"passed": 0,
"failed": 0,
"scenarios": []
}
```

```
for scenario in self.scenarios:
result = await self.run_scenario(scenario)
all_results["scenarios"].append(result)
```

```
if result["success"]:
all_results["passed"] += 1
else:
all_results["failed"] += 1
```

```
return all_results
```

DR Test Scenarios

```
class StandardDRTests:
```

```
    def __init__(
        self,
        db_failover: DatabaseFailoverManager,
        backup_manager: MultiTierBackupManager,
        failover_controller: FailoverController
    ):
        self.db_failover = db_failover
        self.backup_manager = backup_manager
        self.failover_controller = failover_controller
```

```
    def create_database_failover_test(self) -> DRTestScenario:
        scenario = DRTestScenario(
            name="database_failover",
            description="Test automatic database failover to replica"
        )
```

```
    async def verify_primary():
        assert await self.db_failover.check_primary_health()
```

```
    async def simulate_failure():
        pass
```

```
    async def verify_failover():
        result = await self.db_failover.automated_failover()
        assert result["success"]
```

```
    async def verify_application():
        pass
```

```
    scenario.add_step("verify_primary_healthy", verify_primary)
    scenario.add_step("simulate_primary_failure", simulate_failure)
    scenario.add_step("verify_automatic_failover", verify_failover)
    scenario.add_step("verify_application_connectivity",
```

```
verify_application)
```

```
return scenario
```

```
def create_backup_restore_test(self) -> DRTestScenario:  
    scenario = DRTestScenario(  
        name="backup_restore",  
        description="Test backup creation and restoration"  
    )
```

```
    async def create_test_backup():  
        test_data = b"test backup data"  
        result = self.backup_manager.upload_backup(  
            test_data,  
            "dr_test",  
            "hot"  
        )  
        assert result["key"]
```

```
    async def verify_backup_integrity():  
        pass
```

```
    async def test_restore():  
        pass
```

```
    scenario.add_step("create_test_backup", create_test_backup)  
    scenario.add_step("verify_backup_integrity",  
        verify_backup_integrity)  
    scenario.add_step("test_restore_procedure", test_restore)
```

```
return scenario
```

```
def create_region_failover_test(self) -> DRTestScenario:  
    scenario = DRTestScenario(  
        name="region_failover",  
        description="Test failover to secondary region"  
    )
```

```
    async def verify_primary_region():  
        pass
```

```
async def initiate_failover():  
    pass
```

```
async def verify_secondary_region():  
    pass
```

```
async def verify_data_consistency():  
    pass
```

```
scenario.add_step("verify_primary_region", verify_primary_region)  
scenario.add_step("initiate_region_failover", initiate_failover)  
scenario.add_step("verify_secondary_region",  
verify_secondary_region)  
scenario.add_step("verify_data_consistency",  
verify_data_consistency)
```

```
return scenario
```

SECTION 6: RECOVERY RUNBOOKS

Recovery Procedures

```
class RecoveryRunbooks:
```

```
    @staticmethod  
    def complete_system_recovery() -> str:  
        return ""
```

COMPLETE SYSTEM RECOVERY PROCEDURE

PHASE 1: ASSESSMENT (15 minutes)

1. Identify scope of disaster
 - Which systems are affected?
 - What data may be lost?
 - What is the current state of backups?
2. Activate incident response team
 - Notify on-call engineers

- Establish communication channel
- Assign roles (incident commander, communications, technical lead)

3. Document timeline

- When did the incident start?
- What was the last known good state?
- What recovery point is acceptable?

PHASE 2: INFRASTRUCTURE RECOVERY (30-60 minutes)

1. Provision new infrastructure

terraform init

terraform plan -var-file = disaster-recovery.tfvars

terraform apply -auto-approve

2. Restore networking

- Verify VPC and subnets
- Configure security groups
- Set up load balancers

3. Deploy base services

kubectl apply -f infrastructure/

kubectl wait --for=condition=ready pod -l app=core-services

PHASE 3: DATA RECOVERY (60-120 minutes)

1. Identify latest valid backup

aws s3 ls s3://backups/hot/ --recursive | sort | tail -10

2. Restore database

- Download backup
- Restore to new database instance
- Verify data integrity

3. Restore application data

- Configuration files
- Encryption keys (from secure storage)

- User uploaded files

PHASE 4: APPLICATION RECOVERY (30-60 minutes)

1. Deploy applications
kubectl apply -f applications/
kubectl rollout status deployment/app
2. Verify connectivity
 - Database connections
 - External API access
 - Internal service mesh
3. Run smoke tests
pytest tests/smoke/ -v

PHASE 5: VALIDATION (30 minutes)

1. Verify critical functionality
 - User authentication
 - Transaction processing
 - Telegram bot responses
2. Check data consistency
 - Compare record counts
 - Verify recent transactions
 - Check wallet balances
3. Monitor for errors
 - Application logs
 - Error rates
 - Performance metrics

PHASE 6: COMMUNICATION (Ongoing)

1. Internal updates
 - Status to leadership every 30 minutes

- Technical updates to engineering team

2. External communication

- Status page updates
- User notifications
- Social media updates

3. Post-incident

- Timeline documentation
- Root cause analysis
- Improvement actions

"""

@staticmethod

def wallet_emergency_recovery() -> str:

return """

WALLET EMERGENCY RECOVERY PROCEDURE

CRITICAL: This procedure handles recovery of cryptocurrency wallets.

Mistakes can result in permanent loss of funds.

STEP 1: SECURE THE ENVIRONMENT

1. Use air-gapped machine if possible
2. Disable network connectivity during key handling
3. Ensure no screen recording or keyloggers active
4. Work in private, secure location

STEP 2: LOCATE BACKUP FILES

1. Check primary backup location
ls -la /secure/backups/wallets/
2. Check secondary backup location
aws s3 ls s3://secure-backups/wallets/
3. Verify backup integrity before proceeding

```
shasum -a 256 wallet_backup.enc
```

STEP 3: DECRYPT WALLET BACKUP

1. Load decryption key from secure storage
2. Decrypt backup file

```
python decrypt_wallet.py --input wallet_backup.enc --output wallet.json
```
3. Verify decrypted data structure

STEP 4: VERIFY WALLET CONTENTS

1. Check addresses are valid
2. Verify private keys correspond to addresses
3. Do NOT broadcast any transactions yet

STEP 5: SECURE FUND TRANSFER (if compromised)

1. Generate new wallet addresses
2. Prepare transfer transactions
3. Review transaction details carefully
4. Broadcast transactions
5. Monitor for confirmation

STEP 6: UPDATE SYSTEMS

1. Update application with new wallet addresses
2. Rotate all related API keys and secrets
3. Update monitoring for new addresses

STEP 7: POST-RECOVERY

1. Secure new wallet backup
2. Document recovery process

3. Investigate root cause
 4. Update security procedures
- """

```
@staticmethod
def telegram_bot_recovery() -> str:
    return """
```

TELEGRAM BOT RECOVERY PROCEDURE

STEP 1: VERIFY BOT STATUS

1. Check bot token validity
`curl https://api.telegram.org/bot<TOKEN>/getMe`
2. Check webhook status
`curl https://api.telegram.org/bot<TOKEN>/getWebhookInfo`
3. Review recent updates
`curl https://api.telegram.org/bot<TOKEN>/getUpdates`

STEP 2: RESET WEBHOOK

1. Delete current webhook
`curl -X POST https://api.telegram.org/bot<TOKEN>/deleteWebhook`
2. Set new webhook
`curl -X POST https://api.telegram.org/bot<TOKEN>/setWebhook`
\\
`-d "url=https://your-domain.com/webhook" \\`
`-d "secret_token=YOUR_SECRET"`

STEP 3: DEPLOY BOT APPLICATION

1. Deploy bot service
`kubectl apply -f telegram-bot/deployment.yaml`
2. Verify pod is running

```
kubectl get pods -l app=telegram-bot
```

3. Check logs for errors

```
kubectl logs -l app=telegram-bot --tail=100
```

STEP 4: VERIFY FUNCTIONALITY

1. Send test message to bot
2. Verify response received
3. Test critical commands
4. Check database connectivity

STEP 5: RESTORE USER SESSIONS

1. Check session data backup
2. Restore user conversation states
3. Notify users of service restoration

""""

This chapter covered disaster recovery planning and procedures. The next chapter provides the final summary and deployment checklist.

BOOK 8: PRODUCTION DEPLOYMENT

Chapter 8: Deployment Checklist and Summary

This final chapter consolidates everything from the entire book series into actionable checklists. Use these as your pre-flight checks before launching any crypto or Telegram application to production.

SECTION 1: PRE-DEPLOYMENT CHECKLIST

Infrastructure Readiness

```
class InfrastructureChecklist:
```

```
    def __init__(self):
```

```

self.checks = {
    "compute": [
        ("Container orchestration configured", False),
        ("Auto-scaling policies defined", False),
        ("Resource limits set for all containers", False),
        ("Health checks configured", False),
        ("Graceful shutdown handlers implemented", False)
    ],
    "networking": [
        ("Load balancer configured", False),
        ("SSL/TLS certificates installed", False),
        ("DNS records configured", False),
        ("Firewall rules reviewed", False),
        ("VPN access for internal services", False)
    ],
    "storage": [
        ("Database provisioned with replication", False),
        ("Redis cluster configured", False),
        ("File storage with CDN", False),
        ("Backup storage in separate region", False),
        ("Storage encryption enabled", False)
    ],
    "monitoring": [
        ("Metrics collection configured", False),
        ("Log aggregation set up", False),
        ("Alerting rules defined", False),
        ("Dashboards created", False),
        ("On-call rotation established", False)
    ]
}

```

```

def run_checks(self) -> Dict:
    results = {}
    total_passed = 0
    total_checks = 0

```

```

for category, items in self.checks.items():
    category_results = []
    for check_name, status in items:
        category_results.append({

```

```

"check": check_name,
"passed": status
})
total_checks += 1
if status:
    total_passed += 1

results[category] = category_results

results["summary"] = {
    "total": total_checks,
    "passed": total_passed,
    "failed": total_checks - total_passed,
    "ready": total_passed == total_checks
}

return results

```

Security Verification

```
class SecurityChecklist:
```

```

    def __init__(self):
        self.checks = {
            "authentication": [
                ("Multi-factor authentication available", False),
                ("Session management secure", False),
                ("Password hashing using bcrypt/argon2", False),
                ("JWT tokens with short expiry", False),
                ("Account lockout after failed attempts", False)
            ],
            "authorization": [
                ("Role-based access control implemented", False),
                ("Principle of least privilege applied", False),
                ("API endpoints require authentication", False),
                ("Admin functions protected", False),
                ("Telegram user verification", False)
            ],
            "data_protection": [

```

```
(
    ("Encryption at rest enabled", False),
    ("Encryption in transit enforced", False),
    ("Private keys stored securely", False),
    ("Secrets not in code or logs", False),
    ("PII handling compliant", False)
),
"application_security": [
    ("Input validation on all endpoints", False),
    ("SQL injection prevention", False),
    ("XSS protection headers", False),
    ("CSRF protection enabled", False),
    ("Rate limiting configured", False)
],
"crypto_security": [
    ("Hot wallet limits enforced", False),
    ("Multi-signature for large transactions", False),
    ("Transaction signing isolated", False),
    ("Address validation implemented", False),
    ("Withdrawal delays for security", False)
]
}
```

```
def generate_report(self) -> str:
    report_lines = ["SECURITY CHECKLIST REPORT", "=" * 50, ""]

    for category, items in self.checks.items():
        report_lines.append(f"\n{category.upper().replace('_', ' ')}")
        report_lines.append("-" * 40)

        for check_name, status in items:
            status_str = "[PASS]" if status else "[FAIL]"
            report_lines.append(f" {status_str} {check_name}")

    return "\n".join(report_lines)
```

SECTION 2: APPLICATION CHECKLIST

Code Quality Verification


```
class CodeQualityChecklist:
```

```
    def __init__(self):
        self.checks = {
            "testing": [
                ("Unit test coverage above 80%", False),
                ("Integration tests passing", False),
                ("End-to-end tests passing", False),
                ("Security tests passing", False),
                ("Performance tests baseline established", False)
            ],
            "code_review": [
                ("All code reviewed by peers", False),
                ("Security-sensitive code reviewed by security team", False),
                ("Smart contracts audited", False),
                ("No known vulnerabilities in dependencies", False),
                ("Static analysis passing", False)
            ],
            "documentation": [
                ("API documentation complete", False),
                ("Deployment runbook created", False),
                ("Operational procedures documented", False),
                ("Architecture diagrams updated", False),
                ("Incident response plan documented", False)
            ],
            "configuration": [
                ("Environment variables documented", False),
                ("Feature flags configured", False),
                ("Logging levels appropriate", False),
                ("Debug mode disabled", False),
                ("Production configuration validated", False)
            ]
        }
```

Smart Contract Verification

```
class SmartContractChecklist:
```

```
    def __init__(self):
```

```

self.checks = {
"security_audit": [
("External audit completed", False),
("All findings addressed", False),
("Reentrancy protection verified", False),
("Access control reviewed", False),
("Integer overflow protection", False)
],
"testing": [
("Unit tests comprehensive", False),
("Fuzz testing performed", False),
("Invariant testing passing", False),
("Fork testing on mainnet data", False),
("Gas optimization verified", False)
],
"deployment_prep": [
("Constructor parameters verified", False),
("Deployment script tested on testnet", False),
("Contract verification on explorer", False),
("Admin keys secured", False),
("Upgrade mechanism tested", False)
],
"monitoring": [
("Event monitoring configured", False),
("Balance alerts set up", False),
("Transaction monitoring active", False),
("Anomaly detection enabled", False),
("Emergency pause tested", False)
]
}

```

SECTION 3: TELEGRAM BOT CHECKLIST

Bot Configuration Verification

```

class TelegramBotChecklist:

```

```

    def __init__(self):
        self.checks = {

```

```

"bot_setup": [
    ("Bot token secured", False),
    ("Webhook configured with HTTPS", False),
    ("Webhook secret token set", False),
    ("Bot commands registered", False),
    ("Bot description updated", False)
],
"functionality": [
    ("All commands tested", False),
    ("Inline keyboard handling", False),
    ("Callback query handling", False),
    ("Error messages user-friendly", False),
    ("Rate limiting per user", False)
],
"security": [
    ("User ID validation", False),
    ("Admin commands restricted", False),
    ("Input sanitization", False),
    ("Transaction confirmation required", False),
    ("Session timeout implemented", False)
],
"reliability": [
    ("Webhook retry handling", False),
    ("Database connection pooling", False),
    ("Graceful error handling", False),
    ("User state persistence", False),
    ("Message queue for high load", False)
]
}

```

SECTION 4: OPERATIONAL READINESS

Team Readiness

```
class OperationalReadinessChecklist:
```

```

    def __init__(self):
        self.checks = {
            "team": [

```

```

("On-call rotation established", False),
("Escalation paths defined", False),
("Contact information updated", False),
("Access credentials distributed", False),
("Training completed", False)
],
"procedures": [
("Deployment procedure documented", False),
("Rollback procedure tested", False),
("Incident response plan ready", False),
("Communication templates prepared", False),
("Post-incident review process defined", False)
],
"tools": [
("Monitoring dashboards accessible", False),
("Log aggregation queryable", False),
("Alerting notifications working", False),
("Deployment pipeline functional", False),
("Secret management accessible", False)
],
"communication": [
("Status page configured", False),
("User notification channels ready", False),
("Internal communication channel set up", False),
("External stakeholder contacts updated", False),
("Social media accounts accessible", False)
]
}

```

Disaster Recovery Readiness

```
class DisasterRecoveryChecklist:
```

```

    def __init__(self):
        self.checks = {
            "backups": [
                ("Database backups automated", False),
                ("Backup restoration tested", False),
                ("Cross-region replication active", False),

```

```

("Backup encryption verified", False),
("Backup retention policy configured", False)
],
"failover": [
("Database failover tested", False),
("Application failover tested", False),
("DNS failover configured", False),
("Load balancer health checks", False),
("Recovery time objectives met", False)
],
"recovery": [
("Recovery runbooks documented", False),
("Recovery procedures tested", False),
("Data recovery procedures tested", False),
("Wallet recovery procedure verified", False),
("Communication plan tested", False)
]
}

```

SECTION 5: DEPLOYMENT PROCEDURE

Pre-Deployment Steps

```
class DeploymentProcedure:
```

```

    @staticmethod
    def pre_deployment_steps() -> str:
        return """

```

PRE-DEPLOYMENT CHECKLIST

1. Code Freeze

- All code changes merged to main branch
- No pending pull requests for this release
- Version number updated
- Changelog updated

2. Testing Verification

- All automated tests passing
- Manual testing completed

- Performance testing baseline met
- Security scan completed

3. Infrastructure Preparation

- Resources scaled appropriately
- Database migrations prepared
- Configuration updates ready
- Secrets rotated if needed

4. Communication

- Team notified of deployment window
- Status page updated
- Users notified if downtime expected
- Support team briefed

5. Backup Verification

- Recent backup available
- Backup restoration tested
- Rollback plan documented
- Previous version tagged

"""

@staticmethod

def deployment_steps() -> str:

return """

DEPLOYMENT PROCEDURE

1. Pre-Flight Checks

kubectl get nodes

kubectl get pods -n production

Check all systems healthy

2. Create Backup

pg_dump production_db > pre_deploy_backup.sql

Tag current deployment version

3. Apply Database Migrations

alembic upgrade head

Verify migration success

Test critical queries

4. Deploy Application

kubectl set image deployment/app app = app:new-version

kubectl rollout status deployment/app

Watch for pod startup errors

5. Verify Deployment

Run smoke tests

Check application logs

Verify metrics collection

Test critical user flows

6. Monitor

Watch error rates for 30 minutes

Monitor response times

Check resource utilization

Verify no memory leaks

7. Post-Deployment

Update deployment log

Notify team of completion

Update status page

Archive deployment artifacts

"""

@staticmethod

def rollback_steps() -> str:

return """

ROLLBACK PROCEDURE

TRIGGER ROLLBACK IF:

- Error rate exceeds 5% for 5 minutes
- Response time doubles for 10 minutes
- Critical functionality broken
- Security vulnerability discovered

ROLLBACK STEPS:

1. Announce Rollback

Notify team immediately

Update status page

2. Revert Application

kubectl rollout undo deployment/app

kubectl rollout status deployment/app

Verify pods running previous version

3. Revert Database (if needed)

alembic downgrade -1

Verify data integrity

4. Verify Rollback

Run smoke tests

Check error rates

Verify user flows working

5. Post-Rollback

Document what went wrong

Create incident report

Plan fixes for next attempt

""""

SECTION 6: PRODUCTION LAUNCH

Launch Day Procedure

```
class LaunchDayProcedure:
```

```
    @staticmethod
```

```
    def launch_checklist() -> str:
```

```
        return """"
```

PRODUCTION LAUNCH CHECKLIST

T-24 HOURS

[] Final code review complete

[] All tests passing

[] Security audit findings addressed

[] Infrastructure provisioned

[] DNS records configured

- ☐ SSL certificates installed
- ☐ Monitoring configured
- ☐ Alerting tested
- ☐ On-call schedule confirmed
- ☐ Rollback plan documented

T-4 HOURS

- ☐ Team standup and role assignment
- ☐ Communication channels verified
- ☐ Final backup taken
- ☐ Status page prepared
- ☐ Support team briefed

T-1 HOUR

- ☐ Go/no-go decision
- ☐ All team members available
- ☐ Infrastructure health verified
- ☐ Final smoke test passed

LAUNCH

- ☐ Deploy application
- ☐ Verify deployment successful
- ☐ Run smoke tests
- ☐ Enable production traffic
- ☐ Monitor closely for issues

T + 1 HOUR

- ☐ Review metrics
- ☐ Check error logs
- ☐ Verify user feedback
- ☐ Document any issues

T + 24 HOURS

- ☐ Post-launch review
- ☐ Address any issues found
- ☐ Update documentation
- ☐ Celebrate success

|||||

SECTION 7: ONGOING OPERATIONS

Operational Tasks

class OngoingOperationsChecklist:

 @staticmethod

 def daily_checks() -> str:

 return """

DAILY OPERATIONAL CHECKS

1. System Health

- Review overnight alerts
- Check error rates
- Verify backup completion
- Review security logs

2. Application Metrics

- Response time trends
- Error rate trends
- Resource utilization
- User activity patterns

3. Crypto Operations

- Wallet balances
- Pending transactions
- Gas prices
- Network status

4. Telegram Bot

- Message delivery status
- User engagement metrics
- Error responses
- Queue depth

"""

 @staticmethod

 def weekly_checks() -> str:

 return """

WEEKLY OPERATIONAL CHECKS

1. Security Review

- Review access logs
- Check for suspicious activity
- Update dependencies
- Review security alerts

2. Performance Review

- Analyze slow queries
- Review resource trends
- Identify optimization opportunities
- Capacity planning

3. Backup Verification

- Test backup restoration
- Verify backup integrity
- Review retention policies
- Cross-region backup status

4. Documentation

- Update runbooks as needed
- Review incident history
- Update contact information
- Training needs assessment

"""

@staticmethod

def monthly_checks() -> str:

return """

MONTHLY OPERATIONAL CHECKS

1. Disaster Recovery

- Test failover procedures
- Verify recovery time objectives
- Update recovery documentation
- Review DR test results

2. Security Audit

- Vulnerability assessment
- Penetration testing review

- Access rights audit
- Secret rotation

3. Cost Review

- Infrastructure costs
- Transaction fees
- Optimization opportunities
- Budget alignment

4. Compliance Review

- Regulatory requirements
- Audit trail verification
- Policy compliance
- Documentation updates

"""

SECTION 8: SERIES SUMMARY

Complete Knowledge Map

class SeriesSummary:

```
@staticmethod
def book_overview() -> str:
    return """
```

CRYPTO AND TELEGRAM DEVELOPMENT SERIES

BOOK 1: BLOCKCHAIN FUNDAMENTALS

Understanding how blockchains work from first principles. Cryptographic foundations. Consensus mechanisms. Network protocols. Transaction lifecycle. Block structure. Node operations.

BOOK 2: SMART CONTRACT DEVELOPMENT

Solidity programming language. EVM architecture. Contract deployment. Function visibility. State management. Events and logging. Error handling. Gas optimization.

BOOK 3: TOKEN DEVELOPMENT

ERC-20 token standard. Token economics. Minting and burning.

Transfer mechanisms. Approval patterns. Token security. Testing tokens. Deployment strategies.

BOOK 4: DEFI PROTOCOLS

Decentralized exchanges. Automated market makers. Liquidity pools. Yield farming. Flash loans. Price oracles. Protocol integration. Risk management.

BOOK 5: TELEGRAM BOT INTEGRATION

Bot API fundamentals. Webhook configuration. Command handling. Inline keyboards. User management. Session handling. Crypto bot features. Payment integration.

BOOK 6: ADVANCED DEX DEVELOPMENT

Order book implementation. Matching engines. Liquidity aggregation. Cross-chain bridges. MEV protection. Advanced trading features. Protocol governance.

BOOK 7: SECURITY AND AUDITING

Vulnerability patterns. Security auditing process. Static analysis tools. Runtime testing. Formal verification. Incident response. Secure development practices.

BOOK 8: PRODUCTION DEPLOYMENT

Deployment infrastructure. Monitoring and observability. High availability. Scaling strategies. Security hardening. Maintenance operations. Disaster recovery.

"""

@staticmethod

def key_takeaways() -> str:

return """

KEY TAKEAWAYS

SECURITY FIRST

Every decision must consider security implications. In crypto, mistakes are expensive and often irreversible. Always assume attackers are watching.

TEST EVERYTHING

Untested code is broken code. Unit tests. Integration tests. Fuzz tests. Invariant tests. Fork tests. Test in production-like environments.

MONITOR CONSTANTLY

You cannot fix what you cannot see. Metrics, logs, traces, alerts. Know your system's normal behavior so you can detect anomalies.

PLAN FOR FAILURE

Systems fail. Networks partition. Data centers flood. Have backups. Have failover. Have recovery procedures. Test them regularly.

AUTOMATE OPERATIONS

Manual processes are error-prone and unscalable. Automate deployments. Automate testing. Automate monitoring. Automate recovery.

DOCUMENT THOROUGHLY

Future you will not remember why you made that decision. Document architecture. Document procedures. Document incidents. Keep documentation current.

ITERATE CONTINUOUSLY

Launch is not the end. Monitor user behavior. Gather feedback. Fix issues quickly. Improve continuously. Stay ahead of attackers.

""""

Final Verification

class FinalVerification:

```
def __init__(self):
    self.all_checklists = [
        InfrastructureChecklist(),
        SecurityChecklist(),
        CodeQualityChecklist(),
        SmartContractChecklist(),
        TelegramBotChecklist(),
        OperationalReadinessChecklist(),
```

```
DisasterRecoveryChecklist()
```

```
]
```

```
def run_all_checks(self) -> Dict:
```

```
    results = {
```

```
        "checklists": [],
```

```
        "total_checks": 0,
```

```
        "total_passed": 0,
```

```
        "ready_for_production": True
```

```
    }
```

```
    for checklist in self.all_checklists:
```

```
        checklist_name = checklist.__class__.__name__
```

```
        check_count = sum(len(items) for items in  
        checklist.checks.values())
```

```
        passed_count = sum(
```

```
            1 for items in checklist.checks.values()
```

```
            for _, status in items if status
```

```
        )
```

```
    results["checklists"].append({
```

```
        "name": checklist_name,
```

```
        "total": check_count,
```

```
        "passed": passed_count,
```

```
        "complete": passed_count == check_count
```

```
    })
```

```
    results["total_checks"] += check_count
```

```
    results["total_passed"] += passed_count
```

```
    if passed_count != check_count:
```

```
        results["ready_for_production"] = False
```

```
    return results
```

```
def generate_final_report(self) -> str:
```

```
    results = self.run_all_checks()
```

```
    report = []
```

```
    report.append("=" * 60)
```

```

report.append("PRODUCTION READINESS REPORT")
report.append(" " * 60)
report.append("")

for checklist in results["checklists"]:
    status = "READY" if checklist["complete"] else "NOT READY"
    report.append(
        f'{checklist["name"]}: {checklist["passed"]}/{checklist["total"]} '
        f'[{status}]'
    )

report.append("")
report.append("- " * 60)
report.append(
    f'OVERALL: {results["total_passed"]}/{results["total_checks"]} checks '
    f'passed'
)

if results["ready_for_production"]:
    report.append("")
    report.append("STATUS: READY FOR PRODUCTION")
else:
    report.append("")
    report.append("STATUS: NOT READY - Address failed checks '
before deployment")

report.append(" " * 60)

return "\n".join(report)

```

This concludes Book 8: Production Deployment and the complete Crypto and Telegram Development Series.

You now have the knowledge to build, secure, deploy, and operate production-grade cryptocurrency applications with Telegram integration. From blockchain fundamentals to disaster recovery, every aspect has been covered.

Remember: security is not a feature, it is a foundation. Test

everything. Monitor constantly. Plan for failure. Iterate continuously.

Build systems that protect user funds. Build systems that scale. Build systems that recover from disasters. Build systems that you would trust with your own money.

The knowledge is yours. Use it wisely.